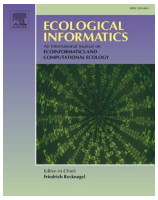
Ecological Informatics 80 (2024) 102486

Contents lists available at ScienceDirect

Ecological Informatics

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



Faster inference from state space models via GPU computing

Calliste Fagard-Jenkin *, Len Thomas

Centre for Research into Ecological and Environmental Modelling and School of Mathematics and Statistics, University of St Andrews, Scotland, UK

ARTICLE INFO

Keywords:
Bayesian inference
CUDA
GPU
Grey seal
Halichoerus grypus
Particle filter
Parallel processing
Particle Markov chain Monte Carlo
Population dynamics model

ABSTRACT

Inexpensive Graphics Processing Units (GPUs) offer the potential to greatly speed up computation by employing their massively parallel architecture to perform arithmetic operations more efficiently. Population dynamics models are important tools in ecology and conservation. Modern Bayesian approaches allow biologically realistic models to be constructed and fitted to multiple data sources in an integrated modelling framework based on a class of statistical models called state space models. However, model fitting is often slow, requiring hours to weeks of computation. We demonstrate the benefits of GPU computing using a model for the population dynamics of British grey seals, fitted with a particle Markov chain Monte Carlo algorithm. Speed-ups of two orders of magnitude were obtained for estimations of the log-likelihood, compared to a traditional 'CPU-only' implementation, allowing for an accurate method of inference to be used where this was previously too computationally expensive to be viable. GPU computing has enormous potential, but one barrier to further adoption is a steep learning curve, due to GPUs' unique hardware architecture. We provide a detailed description of hardware and software setup, and our case study provides a template for other similar applications. We also provide a detailed tutorial-style description of GPU hardware architectures, and examples of important GPU-specific programming practices.

1. Introduction

Graphics Processing Units (GPUs) are specialised pieces of computer hardware designed to facilitate the fast rendering of digital images. This involves repeated applications of the same operation on many different inputs, such as image pixels. To make this as efficient as possible, GPUs are designed to perform many repeats at the same time—i.e., in parallel. Their highly parallel architecture makes them well suited for the fast execution of certain mathematical and statistical tasks, with hundred-fold speed-ups feasible compared with conventional implementations (e.g. Lee et al., 2010b; Suchard et al., 2010). Partly driven by the popularity of video gaming and crypto-currency mining, powerful GPUs have been incorporated into relatively inexpensive consumer-grade products, potentially putting GPU computing within reach of ecologists and evolutionary biologists.

To illustrate the use of GPUs in ecology and evolution, we searched the journal *Ecological Informatics* for articles that mentioned 'GPU' in the text, and reviewed them to determine how they used the hardware. Of the 2150 articles published between 2006 (Volume 1) and 2023 (Volume 76), we found 143 that made clear use of GPUs for scientific computation. The first was in 2016 (Millán et al., 2016), and thereafter the proportion of articles using GPU computing rose rapidly from approximately 1% in 2016 and 2017 to 12–15% in 2021–2023. By far the most common use (139 of 143 articles, i.e., 97%) was in machine learning where the training phase can readily be parallelised and accessible software tools are available to utilise GPUs for this purpose. (A review of machine learning in ecology is given by Pichler and Hartig, 2022.) Typical machine learning applications involved recognition and classification of visual images (e.g. Atila et al., 2021) or acoustic recordings (e.g. Nanni et al., 2020). The most used software tools were those that provide accessible extensions to the Python programming language, such as TensorFlow (Abadi et al., 2015), Keras (Chollet et al., 2015), and PyTorch (Paszke et al., 2019), although alternatives using other programming languages are also used, such as the Deep Learning Toolbox by MathWorks (Beale et al., 2018), cuDNN (Chetlur et al., 2014), and Darknet (Redmon, 2013–2016).

Although GPU computing could potentially benefit other application areas, its use in them is less common, likely because of the lack of general-purpose GPU-accelerated software tools for those applications. Only 4 of the 143 articles reviewed in *Ecological Informatics* involved

* Corresponding author at: Centre for Research into Ecological and Environmental Modelling, The Observatory, Buchanan Gardens, University of St Andrews, St Andrews, Fife, KY16 9LZ, Scotland, UK.
E-mail address: cfj2@st-andrews.ac.uk (C. Fagard-Jenkin).

https://doi.org/10.1016/j.ecoinf.2024.102486
Received 8 June 2023; Received in revised form 16 January 2024; Accepted 17 January 2024
Available online 24 January 2024

uses other than machine learning. One (Wu et al., 2006) used GPUs for their original purpose of image visualization, two (Millán et al., 2016; Strnad et al., 2018) undertook Monte Carlo simulations using custom-written software previously created by some of the same authors, and one (Guo et al., 2022) used newly-created custom GPU-accelerated code for a mapping application involving the fusion of multiple datasets. More widely, there are examples of custom-written software across a range of related fields including geospatial modelling (e.g. Kamewar, 2017; Räss et al., 2019; Sreepathi et al., 2017), environmental modelling (e.g. García-Feal et al., 2018; Hou et al., 2023; Sandric et al., 2019; Vacondio et al., 2014), ecological modelling (e.g. Rubinpur and Toledo, 2022; Weiss, 2013; Welch et al., 2013), evolutionary biology (e.g. Chai et al., 2013; Pratas et al., 2009; Rajapaksa et al., 2019; Zhou et al., 2015), epidemiology (e.g. Galvão Filho et al., 2016; Leonenko et al., 2015; Bisset et al., 2012), medical statistics (e.g. Alsmirat et al., 2017; Eklund et al., 2014; Wang et al., 2016), molecular biology (e.g. Anderson et al., 2008; Zhou et al., 2017) and genetics (e.g. Perez-Wohlfeil et al., 2023; Pütz et al., 2013).

One further application area that can benefit greatly from GPUs is some types of statistical modelling. The evaluation of computationally expensive likelihood functions (which give the probability density of a set of observations, given a model formulation and a set of parameter values) or Monte Carlo methods (which estimate quantities of interest via simulation methods) can be efficiently parallelised when they require many independent replications of the same process or can be broken down into many thousands of independent tasks. For example, randomisation tests (which compare the distributions of two data samples by repeatedly resampling new observations from the joint pool of observations) have been parallelised using GPUs with considerable speedups (Van Hemert and Dickerson, 2011). Uptake of GPU hardware use for these applications has been slower than for machine learning, due to several factors. The first is a required familiarity with lower-level programming languages. Programming languages designed for GPU computing, or programming language extensions compatible with GPUs, necessitate a greater degree of direct memory management to remain efficient. This may be unfamiliar to individuals accustomed to higher-level languages such as R and Python. The second is that GPUs' hardware architecture is different from that of CPUs. Algorithms that are computationally efficient on CPUs may parallelise poorly on GPUs. Therefore custom GPU implementations of new algorithms often requires GPU-specific programming principles that can be time-intensive to learn. We discuss some of these principles throughout this article, and in Appendix B. Finally, there exist fewer libraries for GPU computing that allow simulation of random numbers from non-standard distributions. This is a significant barrier to the implementation of many simulation based methods (such as Monte Carlo methods), which form a large part of the statistical algorithms with high potential for parallelism (Terenin et al., 2019).

The purpose of this article is to encourage wider adoption of consumer-level GPU technology for computer-intensive statistical inference in ecology. Our target audience falls into two categories. First is ecological researchers who are familiar with low-level programming languages (such as Python, C, C++ or Fortran) and who could potentially benefit from the use of GPU programming to accelerate their computations. Second is researchers who are not familiar with such languages but who are working on computationally-expensive ecological problems and wish to ascertain whether their target application could potentially benefit from GPU computing significantly enough to justify the time and resource investment required to produce a GPU implementation. We elucidate which families of algorithms and methods are well suited to GPU-style parallelism. We do this via a case study, fitting a non-linear state space model for the population dynamics of UK grey seals—so far as we are aware, this is the first ecological application of GPU computing to state space models. We provide source code (see Data Availability Statement) that will reduce the initial time investment for producing new custom code of this kind. We also include guidance in Appendix A on the installation and use of GPU related software, both on personal computers and via cloud computing. In Appendix B we provide a tutorial introduction to GPU computing concepts.

The rest of the paper is structured as follows. In Section 2, we introduce state space models (SSMs) and model fitting, the differences between Central Processing Unit (CPU) and GPU architectures, and GPU computing. We then outline, in Section 3, the case study SSM and associated Monte Carlo fitting algorithm based around a technique called particle filtering. We describe a CPU-only implementation and show how code profiling (a form of program analysis that can identify time and memory-intensive aspects of the calculation) can be used to determine how best to parallelise the implementation. We show how GPU-programming practices were used and, in Section 4, we report the reductions in compute time this implementation achieved and we discuss the benefits of using our GPU-based model fitting process, as well as some wider implications of this highly-parallel approach. Lastly, in Section 5 we discuss lessons from the case study, how these lessons may be applied more generally, and the future of GPU computing in ecological statistics.

2. Background

2.1. Inference for state space models

SSMs describe the evolution of two linked stochastic processes in discrete time: one a hidden process that represents the true state of a system, and the other an observed process that depends on the hidden process. SSMs are being increasingly used in ecology to model processes such as animal movement, epidemics and population dynamics—see, e.g., the recent review by Auger-Méthé et al. (2021).

Our case study is a population dynamics application. Population dynamics models are an invaluable tool for wildlife monitoring and conservation, for example allowing inference on population changes over time that are constrained by the known biology of the species. The SSM formulation allows the inclusion of additional information, such as the age and sex of individuals, despite these being unobserved when surveying the population of interest. A seminal paper on SSMs of population dynamics is Buckland et al. (2004) and a full treatment is given by Newman et al. (2014). Here we give a brief summary to motivate the computational algorithm used for inference in the case study.

An SSM can be defined as follows. At each discrete occasion t in a survey, there exist s hidden states $x_{t,1:s} = (x_{t,1}, x_{t,2} \ldots x_{t,s})$ (they are 'hidden' in the sense that their true values are not observed directly). In population dynamics applications, these states often represent the number of individuals in different age, size and/or sex classes. We call the set of all possible values for these states at time t the 'state space' and denote it $\mathscr{X}_t$. The realised value of the hidden state at time t is written x_t. We denote the set of possible values for all hidden states across all time steps by $\mathscr{X}_{1:T}$, where T is the final time step. We use θ to refer to all parameters in the model. The hidden states depend only on their values at the previous time step, via a density $g(x_t|x_{t-1}, \theta)$, called the 'state transition density'. (We note, to avoid confusion, that in some literature, the state transition density is conversely denoted by f, and the observation density by g; also that we use the term 'density' to refer to both probability density (for continuous variables) and probability mass (for discrete variables).) We denote by $\zeta(x_0|\theta)$ the density of the hidden states at the time step preceding the start of the survey.

At each time step t, we make an observation y_t, which is dependent on the hidden states via a probability density function $f(y_t|x_t, \theta)$, called the 'observation density'. The observation y_t is dependent only on the hidden states at the same point in time, and so is said to be 'conditionally independent' (given x_t) of observations at other time points.

With these definitions, we can write the likelihood of the model parameters θ given the observations $y_{1:T}$ as

$$\mathscr{L}(\theta|y_{1:T}) = \int_{\mathscr{X}_0} \zeta(x_0|\theta) \int_{\mathscr{X}_1} \cdots \int_{\mathscr{X}_T} \prod_{t=1}^{T} g(x_t|x_{t-1},\theta) f(y_t|x_t,\theta) dx_T \cdots dx_0. \quad (1)$$

Evaluating this likelihood exactly would require performing $T+1$ nested integrals, each of dimension s. Computationally, this is infeasible, and so using the likelihood for inference (to find maximum likelihood estimates, for example) requires an alternative approach (see Newman et al. (2023) for full discussion).

In the special case where the densities for ζ, f, and g are normal and linear, the Kalman filter offers an analytical solution (Kalman, 1960). However, in ecological applications, normality or linearity assumptions may disagree with species' biology. This can make the resulting model unrealistic, or lead to issues with parameter identifiability (Knape, 2008).

Bayesian inference permits the use of inference algorithms such as Markov chain Monte Carlo (MCMC; see Gelman et al., 1995, for an introduction) in conjunction with data augmentation approaches (Borowska and King, 2022). Here, the model is 'augmented' by sampling from the hidden states $x_{0:T}$ as well as the parameters. In this way, the likelihood is easier to evaluate:

$$\mathscr{L}(\theta, x_{0:T}|y_{1:T}) = \zeta(x_0|\theta) \prod_{t=1}^{T} g(x_t|x_{t-1},\theta) f(y_t|x_t,\theta). \quad (2)$$

Data augmentation methods are effective but are burdened with slow convergence times due to the increased dimension of the space they attempt to explore; Markov chains of this kind often have high posterior correlations in latent states and parameter values, leading to infeasible computation times to obtain effective sample sizes that are sufficient for robust inference (e.g. King, 2011). (Effective sample size, ESS, is a measure of the equivalent number of independent samples from the posterior distribution and is inversely related to Monte Carlo error.) Because of this, more advanced Monte Carlo methods are required.

In this article we focus on a Monte Carlo approach, known as particle filtering, which is one possible alternative to data augmentation. This approach consists in estimating the intractable integrals in Eq. 1 by averaging over the likelihood values of a large number of simulated trajectories through the time-series of hidden states. In this way, algorithms such as the bootstrap particle filter (Gordon et al., 1993) allow inference on parameter values without directly evaluating the state-transition density $g(x_t|x_{t-1})$, using simulation from the density instead.

To highlight the benefit of increasing computational efficiency via the use of GPUs, we perform a Bayesian case study via particle MCMC (pMCMC; Andrieu and Roberts, 2009), the term given to MCMC algorithms that use particle filters to estimate the likelihood. These methods are computationally expensive, and a pMCMC run can require days or weeks to obtain enough samples from the posterior distribution for reasonable inference, even using state-of-the-art hardware (Endo et al., 2019; Kattwinkel and Reichert, 2017). Careful implementation and optimization can reduce run-times by up to an order of magnitude; however, despite continual advances in the efficiency of CPU-only implementations (e.g. Sherlock et al., 2015) and the use of these advances in recent literature (e.g. Finke et al., 2019), CPU-only implementations are often still too slow to be viable without large computing clusters (Šukys and Kattwinkel, 2017), especially in situations where the hidden states at each time step are of high dimension (such as multiple age and sex classes). GPU parallelism is an effective solution to reducing this computational burden (Knape and De Valpine, 2012).

2.2. Graphics processing units

We present here a brief overview of the most significant differences between GPU and CPU hardware, and provide an introduction to the basic programming concepts required to utilise GPUs for population dynamics modelling and general purpose GPU programming more widely. More detailed expositions of efficient GPU programming concepts are given by Chopp (2019), Tuomanen (2018), Sanders and Kandrot (2010), Cook (2012) and Farber (2011).

The layout of a CPU, the hardware that executes all commands in 'traditional' computing, can be represented by a simplified example layout (Fig. 1 (a)). The 'cores', that perform the calculations are managed by control structures. These are responsible for delegating tasks, scheduling which instructions are passed to which cores, and managing memory transfers. Caches on the chip provide fast memory, which allow cores to access small amounts of data quickly for efficient calculation, with slower Random Access Memory (RAM) available elsewhere on the motherboard to store larger quantities of information that need to be accessed less frequently. Due to the wide array of tasks expected of a modern-day computer, minimising the time it takes for a given task to terminate requires dedicating a large amount of the CPU's resources to control structures. CPUs typically have a higher clock-speed (a measure related to the number of operations the chip can perform each second) than GPUs, reducing the time they take to perform an individual task.

In situations where flexibility is less important, the total number of calculations performed per unit of time (computational throughput) can be increased by dedicating a higher proportion of resources to compute cores. This is the underlying motivation behind the GPU (Fig. 1 (b)). A GPU chip is comprised of multiple Streaming Multiprocessors (SM) (Fig. 1 (c)). Sacrificing the resources available to the control structures constrains the GPU into the SIMD framework (single instruction multiple data), which only allows processes to run in parallel if they are performing identical tasks. Therefore, while CPUs are faster at performing individual tasks, GPUs will provide higher throughput for scientific computing applications that involve large numbers of independent identical tasks, on different inputs. For example, resampling from a given set of observations many times, to produce a non-parametric bootstrap distribution of a statistic of interest. A further constraint of GPUs is that they typically dedicate fewer resources than CPUs to performing double-precision floating point arithmetic. This has consequences on numerical stability that will be discussed further in following sections.

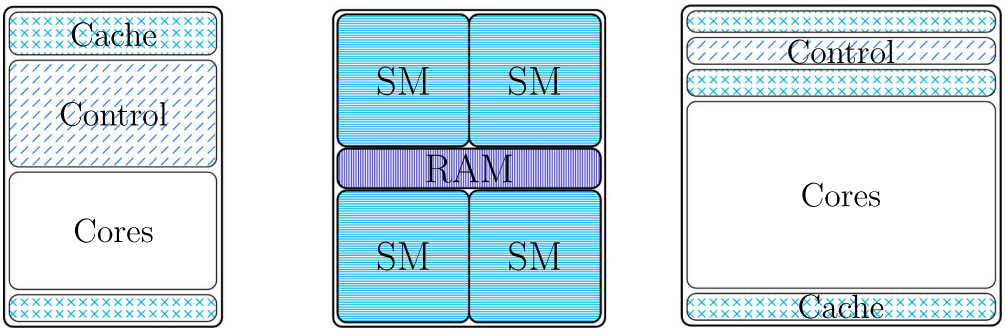
At the time of writing there are two companies that produce popular consumer-grade GPUs: Nvidia and AMD. The Compute Unified Device Architecture (CUDA) (NVIDIA et al., 2020), a proprietary framework for Nvidia GPUs, is a useful tool for creating custom parallel GPU functions, or 'kernels' in CUDA terminology. Due to the popularity of CUDA within the scientific computing community, its integration with commonly used programming languages in computationally intensive statistics and ecological modelling (such as C++ and Fortran), and its high performance for such applications, we have selected it for our case study. The cross-platform OpenCL programming language (Stone et al., 2010) is an alternative that allows for the use of GPUs produced by other manufacturers, such as AMD. Although syntax and terminology vary across CUDA and OpenCL, the programming concepts and algorithm design choices required for efficient implementations are shared between both platforms.

In CUDA, the independent processes in a parallel calculation, which are able to take place simultaneously, are known as 'threads', and are grouped into 'warps' of 32. Threads in a warp execute in lockstep, forcing them to each perform the same task at the same time. Warps can be packaged together into 'blocks' that will be scheduled for execution on the same SM. A more detailed introduction to CUDA terminology is included in Appendix B.

3. Materials and methods

3.1. Grey seal population dynamics model

The case study is a Bayesian state-space model for the regional population dynamics of British grey seals (*Halichoerus Grypus*) that is used in a management context (e.g. Special Committee on Seals, 2021)



(a) Central Processing Unit (CPU) (b) Graphics Processing Unit (GPU) (c) Streaming Multiprocessor (SM)

Fig. 1. Example layouts of a CPU and GPU. Displays the approximate locations and proportion of chip surface area used by cache memory (crosses), control structures (diagonal lines), compute cores (no fill), random access memory (vertical lines) and streaming multiprocessors (horizontal lines). Compute cores perform computations. Cache memory allows cores to access information quickly. Control structures allow for faster computation by efficiently managing cores and memory usage. Streaming multiprocessors (SM) are used to provide cache and control structures to groups of cores.

to estimate total population size and trend as well as demographic parameters. The model, data, prior distributions and a fitting method based on particle filtering are described by Thomas et al. (2019); a summary is given here.

The data comprise estimates of the number of pups born ('pup production') each year between 1984 and 2010 in four regions of Scotland (North Sea NS, Inner Hebrides IH, Outer Hebrides OH and Orkney OR; Fig. 2), along with an independent estimate of the total adult population size in 2008 (94,390 animals, SD 978).

The population dynamics model is structured by age, splitting animals into seven groups: those below one year of age (pups), two years of age, and so forth, until the final group which contains all individuals aged 6 or higher. After the pup stage, the model tracks only females, and animals age 1 or older are referred to as "adults". Hence the total adult female population size in a given year t is $P_t = \sum_{r=1}^{4}\sum_{a=2}^{7} x_{t,r,a}$, where $x_{t,r,a}$ denotes the number of seals in age group a in region r at time t, with $t = 0$ corresponding to 1984.

Under the model, adults survive to the next year with a fixed probability ϕ_a. Pup survival is assumed to be density dependent: higher regional pup production in the previous breeding season ($x_{t-1,r,1}$) leads to lower pup survival probability ($\phi_{t,r}$) according to the equation

$$\phi_{t,r} = \frac{\phi_{pmax}}{1 + \left(\beta_r x_{t-1,r,1}\right)^{\rho}} \tag{3}$$

where ϕ_{pmax} is maximum survival probability (when $x_{t-1,r,1} = 0$), ρ determines the shape of the density dependence relationship and β_r is related to the region-specific carrying capacity (see below). $\frac{100}{\omega}$% of the surviving pups are expected to be female. Following the survival process, all of the individuals in the population increment their age by one year, and each female of breeding age (the final age group only) has a pup with probability α, the fecundity parameter.

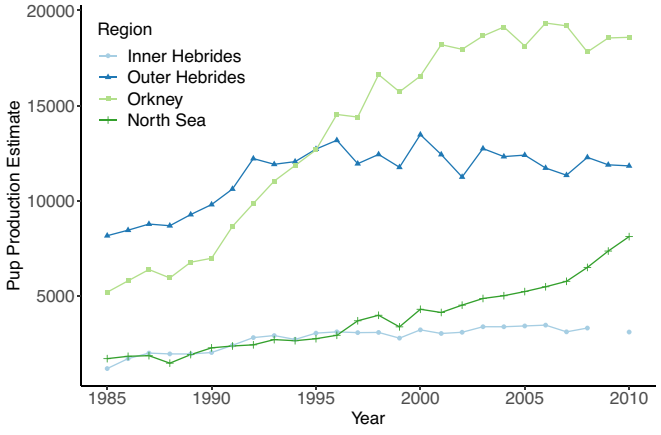


Fig. 2. Annual grey seal pup production estimates in four regions.

In this model, all of the $x_{1:26,1:4,1:7}$ true age group counts are hidden states. We observe a number of pups $y_{t,r}$ for each region and time interval, with some normally distributed error relative to the true pup production value, such that:

$$y_{t,r} \sim \mathcal{N}\left(x_{t,r,1}, \frac{x_{t,r,1}^2}{\psi}\right) \tag{4}$$

where ψ is an observation precision parameter whose value is estimated during the model fitting process.

In 2008, two time points before the end of the survey, we also include information provided by an independent estimate of the total adult population size by Russell et al. (2016). This total population size is assumed to have a gamma distribution with known parameters. An estimate for the number of adults present in the survey population in 2008 (denoted by N_{24}, the total count of individuals in year 24) can by obtained by scaling the number of adult females with a parameter ω, the ratio of the total population size to the number of females:

$$N_{24} = \omega \sum_{r=1}^{4}\sum_{a=2}^{7} x_{24,r,a}; N_{24} - k_0 \sim \text{Gamma}(k_1, k_2) \tag{5}$$

with shift ($k_0 = 59170$), shape ($k_1 = 12.96$), and rate ($k_2 = 2719$) parameters.

The state transition density $g(x_t|x_{t-1}, \theta)$ in this model cannot be evaluated directly. However, we are able to produce random draws from this density by simulating from the sequence of survival processes defined above. The number of one year old seals in a region at time $t+1$ ($x_{t+1,r,2}$) can be obtained with a binomial draw from the number of pups in the previous time step ($x_{t,r,1}$) with probability $\frac{\phi_{t,r}}{\omega}$ using eq. 3. The division by ω ensures that only female pups are recruited into following age stages. In the same way, for adult age groups, the state $x_{t+1,r,a}$ is obtained by simulating the number of survivals from the previous time step and age group $x_{t,r,a-1}$ via a binomial draw with probability ϕ_a. In the case of breeding age females ($x_{t+1,r,7}$) the number of individuals in the following time step is the sum of survivals from both breeding age females ($x_{t,r,7}$) and females aged five ($x_{t,r,6}$). The number of pups born in a given year ($x_{t+1,r,1}$) is a binomial draw from the number of surviving females ($x_{t+1,r,7}$) with probability α, the fecundity. Random samples from the state transition density are sufficient to produce evaluations of the likelihood. This will be discussed in more detail later in this Section.

The model is Bayesian, and hence prior distributions are required for all parameters and the initial states $x_{0,1:4,1:7}$. Distributions used are given in Table 1; full justifications for these distributions are provided by Thomas et al. (2019). To aid specification of the four regional carrying capacity-related parameters, β_r, the prior distributions are set instead on regional carrying capacities, χ_r, where

$$\chi_r = \frac{1}{\beta_r}\left(\frac{\alpha \cdot \phi_{pmax} \cdot \phi_a^5}{2(1-\phi_a)-1}\right)^{\frac{1}{\rho}}. \quad (6)$$

3.2. Auxiliary particle filter approach

To obtain weighted samples from the posterior distribution Thomas et al. (2019) used a particle filtering (also called sequential Monte Carlo) algorithm. Particle filtering employs a large number of simulated samples generated from the model, each representing a possible state of the system, and assigns weights to these samples based on their likelihood of producing the observed data. This process begins by first drawing a large number of parameter values from the prior distribution, along with initial values for each hidden state. Each of these samples is known as a particle, and is given an equal starting weight. Each particle's hidden states are then projected forwards in time by simulating from the population dynamics model, and its weight is updated using the log-likelihood of the observed pup production estimate, given the particle's value for the true count (the hidden state). Once this process is completed for all time steps, particle weights can be used to produce weighted means of the posterior parameter values, or any other statistic of interest.

Particle filtering methods can be inefficient, as the initially proposed sets of parameter values do not necessarily produce particles with high weights. To remedy this, Thomas et al. (2019) adopted rejection control and kernel smoothing methods described by Liu and West (2001), within a tempered auxiliary particle filter (Pitt and Shephard, 1999). We do not describe this method in detail, as we did not use it for model fitting. As noted by Thomas et al. (2019), the kernel smoothing, which consists in adding a small jitter to parameter values tied to resampled particles to avoid creating identical copies, leads to biased samples from the posterior. To minimize this bias, Thomas et al. (2019) used a very small amount of kernel smoothing (discount parameter set to 0.99997 where a value of 1.0 results in no kernel smoothing), but this meant a large number of particles were required for adequate accuracy. Thomas et al. (2019) combined results from 4000 replicate runs each of 1,000,000 particles; this took 72 h to run in parallel batches of 20 on 2 multi-core server-class machines using CPU code written in ANSI C, plus a few hours of post-processing time to combine the runs. We compare the estimates obtained from the GPU algorithm described below with the results they obtained.

Table 1
Summary statistics for the grey seal model prior and posterior distributions. Posterior distributions obtained via auxiliary particle filter (APF) are those produced by Thomas et al. (2019). Abbreviations: SD denotes standard deviation; Be and Ga denote beta and gamma distributions respectively; cc, carrying capacity; DD, density dependent; NS, IH, OH, and Ork denote the North Sea, Inner Hebrides, Outer Hebrides, and Orkney regions.

Parameter	Prior distribution	Prior mean (SD)	Posterior mean via APF (SD)	Posterior mean via pMCMC (SD)
Adult survival ϕ_a	$0.8 + 0.17 \times$ Be(1.6, 1.2)	0.90 (0.04)	0.95 (0.01)	0.95 (0.01)
Pup survival ϕ_{pmax}	Be(2.87, 1.78)	0.62 (0.20)	0.48 (0.09)	0.48 (0.09)
Fecundity α	$0.6 + 0.4 \times$ Be(2, 1.5)	0.83 (0.09)	0.90 (0.06)	0.89 (0.06)
DD shape ρ	Ga(4, 2.5)	10 (5)	5.95 (1.73)	5.62 (0.75)
NS pups at cc χ_{NS}	Ga(4, 5000)	20000 (10000)	15500 (8210)	17600 (10200)
IH pups at cc χ_{IH}	Ga(4, 1250)	5000 (2500)	3110 (173)	3080 (87)
OH pups at cc χ_{OH}	Ga(4, 3750)	15000 (7500)	11700 (535)	11800 (257)
Ork pups at cc χ_{OR}	Ga(4, 10000)	40000 (20000)	17800 (1680)	17800 (786)
Observation precision ψ	Ga(2.1, 66.67)	140 (96.61)	112 (34.6)	132 (17.2)
Sex ratio ω	$1.6 +$ Ga(28.08, 3.70E − 3)	1.7 (0.02)	1.7 (0.02)	1.7 (0.02)

3.3. Particle MCMC

The bias introduced by kernel smoothing can be avoided by using particle filtering in a different manner. Rather than using the particle filter directly to estimate the posterior distribution of the model parameters, it is instead used as part of an MCMC algorithm. In an MCMC algorithm, samples can be obtained from a chosen distribution by via repeated evaluation of the likelihood function. For a detailed introduction to MCMC methods, see Gelman et al. (1995); Brooks et al. (2011). We used a type of MCMC algorithm called the Metropolis-Hastings algorithm, where new sets of parameters are proposed and then either accepted or rejected sequentially, based on the change to the posterior density. Parameter values with a higher posterior density are always accepted, and points with a lower density are accepted with probability equal to the ratio of the posterior density given proposed vs current parameter values. In this framework, we use the particle filter instead to provide an estimate of the log-likelihood for a given set of parameter values (the 'marginal likelihood'), which allows us to calculate the acceptance probability. Here a simple particle filtering algorithm is used: the bootstrap particle filter (Gordon et al., 1993). The final particle weights can be used to derive an unbiased estimate of the log-likelihood (Del Moral, 1996), and unbiased log-likelihood estimates have been shown to enable samples to be drawn from the exact posterior distribution (Andrieu and Roberts, 2009). These samples can be obtained via the pMCMC Metropolis-Hastings algorithm (Algorithm F.2, Appendix F). The bootstrap filter constitutes the majority of the pMCMC computation time, and so is presented in more detail by Algorithm 1 as a means of identifying which aspects are most appropriate for a highly parallel approach, and which are most in need of optimisation. A generic version of this algorithm that does not contain modifications specific to the case study presented in this article is also provided in Appendix F.

3.4. GPU implementation

Algorithms are typically either 'compute-bound' or 'memory-bound'. In the case of the former, the majority of the compute time is spent performing operations on data. In the latter, more time is spent copying data between locations in memory. The amount of time taken to copy any data from one memory location to another is known as the memory's 'latency'. The amount of data that can be transferred from one location in memory to another in a given unit of time is known as the memory's 'bandwidth' (e.g., cache memory typically has a lower latency than off-chip RAM, although off-chip RAM has higher bandwidth). Memory-bound applications are more difficult to parallelise, as adding more compute cores does not increase memory bandwidth or reduce its latency, which remains the bottleneck of the computation. Compute-bound applications are more easily parallelised, as increasing the number of cores allows each core to perform a given task on a reduced quantity of data. However, as the compute time for each core approaches the latency of the available memory, the benefit of each new added core tends to zero. In some cases, using 'too many' cores can lead to worse performance, due to overheads associated with multi-threading and memory management.

In the case where many CPU cores are available on the same device, such as with modern chips housing up to 128 cores on a single die, CPU-

1: set $LL := 0$
2: **for each** particle, i and region, r **do** sample $\boldsymbol{x}^i_{0,r,1:A} \sim \zeta_r$ from the prior **end for**
3: **for each** particle, i **do** set the log weight $\boldsymbol{w}_i := 0$ **end for**
4: **for** $t \in 1{:}T$ **do**
5: **for each** particle, i **do** resample an index $I_i \sim \propto \exp w_{1:N}$ **end for**
6: **for each** particle, i **do** reset $w_i := 0$ **end for**
7: **for each** region, $r \in 1{:}R$ **do**
8: **for each** particle, i **do**
9: set $\tilde{x}^i_{t-1,r,1:A} := x^{I_i}_{t-1,r,1:A}$
10: generate $x^i_{t,r,1:A} \sim g(x^i_{t,r,1:A}|\tilde{x}^i_{t-1,r,1:A}, \theta)$
11: $w_i \mathrel{+}= \log f(y^i_{t,r,1}|x^i_{t,r,1}, \theta)$
12: **end for**
13: **end for**
14: **if** year t has an independent estimate of population size **then**
15: **for each** particle, i **do** $w_i \mathrel{+}= \log h\left(\sum_{r=1}^{R} x^i_{t,r,2:7}|\xi\right)$ **end for**
16: **end if**
17: $LL \mathrel{+}= \log \sum_{i=1}^{N} (\exp\{w_i - M\}) - \log N + M$
18: **end for**
19: **return** LL;

There are R regions, T years of pup counts, A age groups, and N simulated particles used to estimate the log likelihood LL. The function ζ_r is the prior distribution for the initial states in region r. The function $g\left(x_{t,r,1:A}|\tilde{x}_{t-1,r,1:A}, \theta\right)$ is defined by the sequence of survival, age incrimination, recruitment into the adult population, and birth, that characterised the population dynamics model in the previous section. $f\left(y_{t,r,1}|x_{t,r,1}\right)$ is the density of our observed pup count given the true count, and $h\left(\sum_{r=1}^{R} x_{t,r,2:7}|\theta, \xi\right)$ is the density of the total true female adult population size given known values $\xi = k_0, k_1, k_2$ (Thomas et al., 2019). M is any number approximately the size of the largest log weight, and helps to ensure numerical stability. Lines of pseudo-code within a **for each** loop are not dependent on the result of the same calculation for previous indices, and therefore have the potential to be performed in parallel.

Algorithm 1. Bootstrap Particle Filter for Marginal Log Likelihood Estimation.

parallelism can be achieved with far lower overheads. However, these chips are not yet widely available to most consumers, largely due to cost. Server- or cluster-based parallelism is less efficient because computations become memory-bound more rapidly, as the latency of data transfers is increased. This problem is exacerbated for algorithms that require regular or semi-regular communication between threads of execution. The bootstrap particle filter is an example of this: resampling particle weights from the entire population must occur every time step (line 5 in Algorithm 1). This places strict upper bounds on the speedups that can be achieved without a higher density of compute cores per device, and explains why the 'fine-grain' parallelism offered by GPUs is better adapted to this context. GPUs achieve faster communication by giving all of their cores access to the same on-chip global memory, which typically has higher bandwidth than the memory available to CPUs (the computer's off-chip RAM). In addition to this, cores on the same streaming multiprocessor have access to a shared cache, which allows for very fast communication between small numbers of threads.

To determine how best to parallelise a sequence of calculations, it is important to first understand which broad sections of this computation are most expensive, and whether they are compute-bound or memory-bound. It is also important to understand which elements of the algorithm require communication between threads, and which calculations are necessarily dependent on previous results (perhaps performed by other threads). It is worth noting that many algorithms, including Algorithm 1, comprise elements with different computational complexities. Thus, the proportion of the compute time that is dominated by a given element is dependent on the size of the input (in the case of particle filters: the number of particles and the number of observations).

We implemented the bootstrap particle filter in R (R Core Team, 2021) version 4.2.1 and used the code profiling facilities provided by the profvis package (Chang et al., 2020) to determine the relative allocation of compute time to the tasks within the bootstrap filter. This showed that the vast majority of compute time was spent projecting particles forwards to the following time step (Table 2). The proportions of time spent on each task are likely to change with the state-dimensionality of the application at hand and the amount of interaction between particles (Whiteley et al., 2016). Large state-dimensionalities or particle interaction can lead to a higher proportion of compute time spent on the resampling step in particular. In these cases, there exist alternative resampling methods to reduce the impact of particle interactions on computational cost (e.g. Lee and Whiteley, 2016). Note that implementations of the algorithm in different programming languages, especially compiled ones such as C or Fortran, will vary slightly in the proportion of compute time spent in each task.

Whilst it is difficult to profile the time spent performing memory allocations in R, many useful tools exist to perform this task for GPU implementations (Knobloch and Mohr, 2020). We note that in our example, for GPU implementations of the state projection step (line 10 in Algorithm 1) to remain effective, we must ensure that its computation time remains higher than the data-transfer time of hidden states to and from GPU memory. We discuss methods of achieving this in more detail

Table 2
Proportion of total compute time spent performing the listed tasks in a bootstrap particle filter for marginal log-likelihood estimation of the grey seal population dynamics model using 2^{16} particles per evaluation, implemented in R.

Task	Proportion of compute time (%)
Projection of hidden states to the next time step	89.0
Calculate probability density of observed states	2.91
Produce initial hidden state values	2.54
Resample particle weights	1.54
Other	4.01

during following sections.

The suitability of an algorithm to GPU parallelism depends on the hierarchical structure of its **for** and **for each** loops. Broadly speaking, an algorithm where the parallelism is present in the outermost loops will benefit readily from a coarse grain implementation (i.e., CPU-based parallelism) whereas algorithms where the parellelism is found in inner loops will require fine-grain parallelism with smaller overheads (such as GPU computing) to remain efficient.

In Algorithm 1, the only loop that cannot be parallelised is the process moving forwards through time (algorithm lines 4 through 18), because here the output of one iteration of the loop forms the input to the next. Within a given time step, however, each particle is able to independently produce a resample (given knowledge of the other log weights, in line 5), project its hidden state values forward to the following time step (line 10), and evaluate its observation density (line 11). GPUs do this efficiently with 'kernels', which are functions designed to be executed only on the GPU. These functions are written not with the perspective of the entire algorithm, but with that of a single thread of execution, in our case, a particle. Kernels are efficient to launch, requiring almost no overhead, making it trivial to launch tens of thousands to millions of independent calculations. To illustrate the general structure of one of these kernels, we present the pseudocode for the forwards projection kernel (line 10) in Algorithm 2.

Algorithm 2. Hidden State Projection Kernel.

1: Obtain tid, the unique **T**hread **Id**entifier of the thread.
2: **if** tid $< N$ **then**
3: $\text{seed}_{\text{LM}} := \text{seed}_{GM}^{\text{tid}}$
4: Calculate ϕ_{p} using θ and $x_{t-1,r,1}^{\text{anc}}$(with equation 3)
5: Simulate $x_{t,r,2}^{\text{tid}} \sim B(x_{t-1,r,1}^{\text{anc}}, \frac{\phi_{\text{p}}}{\omega})$ using seed_{LM}
6: **for** a in $3:6$ **do**
7: Simulate $x_{t,r,a}^{\text{tid}} \sim B(x_{t-1,r,a-1}^{\text{anc}}, \phi_a)$ using seed_{LM}
8: **end for**
9: Simulate $x_{t,r,7}^{\text{tid}} \sim B(x_{t-1,r,6}^{\text{anc}}, \phi_a) + B(x_{t-1,r,7}^{\text{anc}}, \phi_a)$ using seed_{LM}
10: Simulate $x_{t,r,1}^{\text{tid}} \sim B(x_{t,r,7}^{\text{anc}}, \alpha)$ using seed_{LM}
11: Update $\text{seed}_{\text{GM}}^{\text{tid}} := \text{seed}_{\text{LM}}$
12: **end if**

N represents the total number of particles. ϕ_p is the pup survival probability in year t given the hidden state $x_{t-1,r,1}^{\text{anc}}$, and the parameters θ. The hidden state $x_{t-1,r,a}^{\text{anc}}$ refers to the ancestor of the particle $x_{t,r,a}^{\text{tid}}$ resampled in line 5 of Algorithm 1. seed_{LM} is a single seed stored in local memory, used to generate pseudo-random numbers, and seed_{GM} is a vector of initial random seeds, stored in global memory. $B(n,p)$ denotes the binomial distribution. The conditional statement on line 2 avoids the program attempting to access unallocated memory if the number of threads launched is greater than the number of desired particles.

In Algorithm 2, a locally stored variable (using registers or shared memory) is distinguished from the version stored in global memory by the subscripts LM, and GM, respectively. By storing variables such as seeds for random number generation locally, we avoid needing to update the value in global memory (which has a high read/write latency) every time we wish to generate a random number.

When the GPU executes kernels such as Algorithm 2, each warp (a group of 32 threads running in parallel) executes each line of code simultaneously. By ensuring there are no conditional branches (such as an **if** statement), we avoid situations in which some threads may sit idle waiting for the rest of their warp to perform the other part of the branch. This idling due to branching is known as 'warp divergence', and algorithms that include heavy branching are less likely to benefit from SIMD parallelism. Also relevant is the total time spent waiting for data to be transferred to and from the host (the computer in which the GPUs are housed), when no other 'useful' calculations can take place on the GPU. In our example, code-profiling revealed this is negligibly small, due to careful implementation choices that force the GPU to perform 'single threaded' tasks, avoiding memory transfers to and from the host computer's RAM. More detailed examples of this approach are given in Appendix B.

3.5. Tuning

The number of particle filters used per estimation of the log-likelihood (denoted by n_f), and the number of particles used by each of these particle filters (denoted by n_p), are quantities that must be tuned for a given application, to obtain high ESSs of the posterior in a computationally efficient manner. Increasing n_f or n_p tends to increase the ESS of MCMC samples using this log-likelihood estimator (by reducing the variance of the estimator), whilst simultaneously reducing the number of samples that can be obtained in a given unit of time (due to the increased computational cost). We considered values of n_f between 1 and 25, and values of n_p between 2^{10} and 2^{20}. In situations where an estimate of the log-likelihood is used to calculate MCMC acceptance probabilities, Doucet et al. (2015) suggest that an estimator of the log-likelihood with a standard deviation slightly larger than 1.0 leads to efficient use of computational resources. Therefore, we excluded combinations of n_f and n_p with standard deviation less than 0.5 or more than 2.0. We also excluded any combinations of n_f and n_p that had GPU occupancy (a measure of the total number of active threads, relative to the theoretical maximum) less than 30%. A grid search over the remaining values yielded a selection of $n_f = 3$ and $n_p = 2^{15}$, using average ESS across all parameters over a pilot MCMC chain of 1000 samples as the selection criterion. For each estimation of the log-likelihood, we ran particle filters with the above tuning values simultaneously on two GPUs, taking the mean likelihood estimate, using a total of $2 \times 3 \times 2^{15} = 196608$ particles per log-likelihood estimate. This produced estimates with a standard deviation of approximately 0.82. These tuning values will vary significantly with the computational complexity of the application and the hardware being used.

A non-isotropic Gaussian random-walk proposal was used for the Metrolopis-Hastings step of the pMCMC algorithm (new sets of parameters are proposed with a normal distribution centred on the previous set of parameters, with a variance-covariance matrix structure that allows for off diagonal entries, i.e. correlation between parameters). The variance-covariance matrix (VCM) of the proposal distribution is a tuning parameter that affects the quality of mixing of the resulting MCMC chain. Adaptive MCMC algorithms use the VCM of samples obtained from the posterior to modify the proposal VCM at regular intervals, allowing it to better approximate the correlation structure of the posterior distribution, often using a tempering schedule(Mueller, 2010; Roberts and Rosenthal, 2009). We used a simpler technique that discards all samples from the posterior obtained from one adaptation of the VCM to the next. We began with a diagonal VCM with unit value entries, and produced an MCMC chain of 1000 samples from the posterior. The number of samples obtained was increased in increments of 1000 until 10,000 samples were obtained in one adaptive phase. Following this, chains with 20,000 and 30,000 samples were used to update the proposal VCM twice more. At each iteration, the sample VCM was used as the updated proposal VCM. No tempering schedule was used, as all previous samples were discarded. See Peters et al. (2010) for a more detailed discussion of possible adaptive schemes for ecological pMCMC.

During the analysis the proposal VCM was held constant and chains were initialised at the sample of the most recent adaptive phase with highest posterior density. Two chains of 10^5 samples from the posterior were obtained. Given that the iterative process ensures the final chains begin in an area of high posterior density, no samples were removed as a 'burn-in'. No posterior samples were removed via thinning, as the resulting file sizes still allow for fast post-processing (Link and Eaton, 2012). Convergence was assessed using the Gelman-Rubin diagnostic. To visualise posterior marginal distributions, Guassian kernel density estimation was used, with bandwidth equal to one fifth of the sample standard deviation of posterior samples for each respective parameter.

3.6. Equipment

The machine used for the analysis was equipped with an Intel i7-6700k CPU (released in 2015), and two Nvidia RTX 3070 GPUs (a desktop card released in 2020) operated with the Nvidia-SMI driver, version 470.161.03. Implementations were written in CUDA C++ (NVIDIA et al., 2020), version 11.4, and compiled with the NVCC compiler, version 10.1.243. An additional computer (a GPU-equipped laptop) was used as part of a benchmarking study, comparing run times across a range of hardware, GPU vs CPU and particle numbers

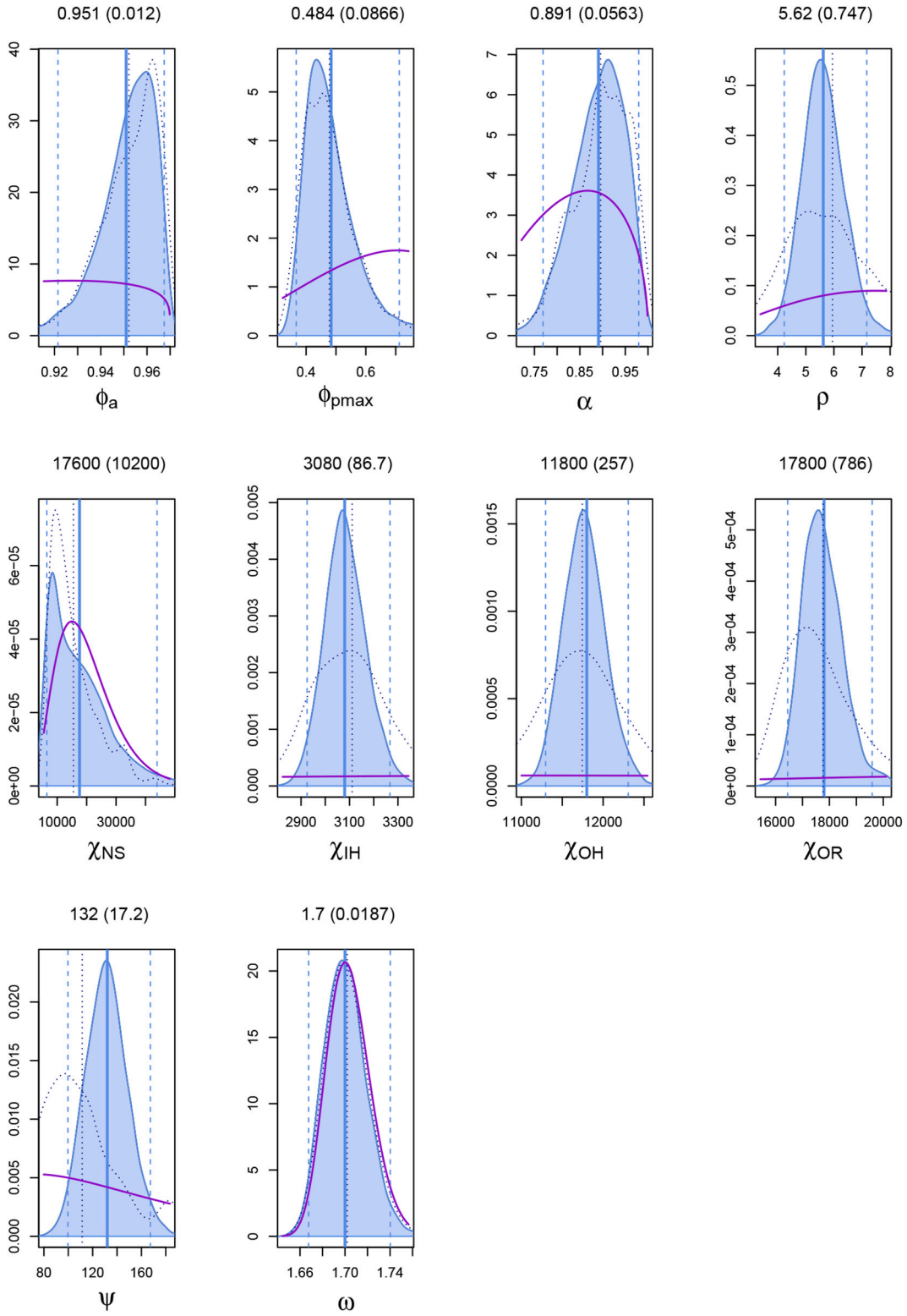


Fig. 3. Posterior marginal distributions for each parameter (shaded under the curve), with 95% credible intervals (vertical dashed lines) and posterior means (solid vertical lines). Prior distributions are given by a solid line with no shading below the curve. Posterior distributions obtained by Thomas et al. (2019) are drawn with dotted lines. Numbers above each plot are the posterior mean and standard deviation (in brackets).

(Appendix C).

4. Results

Trace plots of the MCMC chains displayed good convergence, assessed via Gelman-Rubin diagnostics for each parameter (Fig. D.1 in Appendix D) and trace plots (Fig. D.2 in Appendix D). Mean ESS across model parameters was 1759 samples, ranging from 763 (ϕ_{pmax}) to 6290 (ψ). The MCMC chains took a total of 8435 s to complete (2.34 h).

Posterior marginal distributions for model parameters are shown in Fig. 3 and summarised in Table 1. The posterior mean on adult survival ϕ_a and fecundity α are higher than the prior means, while those on maximum pup survival ϕ_{pmax} and density dependence shape ρ are lower. In all cases there is moderate overlap between prior and posterior distributions, indicating some learning about these parameters from the data (and model structure) but also an influence of the prior distributions. Biplots of each parameter pair indicates strong posterior correlation between ϕ_a and ϕ_{pmax} (Fig. D.3 in Appendix D). For three of the four carrying capacity parameters, χ_{IH}, χ_{OH}, and χ_{OR} the posterior marginal distribution is much narrower than the prior, indicating considerable learning; these correspond with the regions (Inner Hebrides, Outer Hebrides and Orkney) where pup production has levelled off (Fig. 2). For the remaining region, North Sea, the pup production trajectory is still increasing (Fig. 2) and the corresponding carrying capacity parameter χ_{NS} posterior distribution is similar to (but slightly lower than) the prior. The posterior mean on observation precision is similar to the prior but the standard deviation is considerably smaller, indicating moderate learning. The prior and posterior distributions on sex ratio ω are almost identical.

The estimated posterior marginal distributions are generally similar to those obtained by Thomas et al. (2019) (shown as dashed lines in Fig. 3 and summarised in Table 1) using the auxiliary particle filter algorithm. Thomas et al. (2019) reported an ESS size of 478 samples, lower than for any of the parameters in the analysis undertaken here, so we expect more Monte Carlo error in the Thomas results. The distributions for ϕ_a, ϕ_{pmax}, α and ω are nearly identical, although those from Thomas et al. (2019) (except ω) are slightly more 'wiggly' as might be expected from the lower ESS. Estimated posterior means for ρ, χ_{IH}, χ_{OH}, and χ_{OR} are nearly identical but the estimated standard deviation is somewhat higher using the auxiliary particle filter. Posterior mean and standard deviation on χ_{NS} are both somewhat lower using the auxiliary particle filter, while posterior mean on ψ is lower and standard deviation higher.

Speed comparisons between the implementations are necessarily approximate because they used different fitting algorithms (pMCMC vs auxiliary particle filter), computing languages (C++ with CUDA vs C) and computers. Nevertheless, the difference is striking. Thomas et al. (2019) reported an ESS of 478 samples from the posterior, achieved after 72 h of computing on two computers using 20 CPU cores (plus post-processing time, not counted here). Achieving this average ESS via our GPU-accelerated pMCMC implementation would require approximately 40 min, representing a speedup factor of 113.

5. Discussion

5.1. Case study: ecological and modelling implications

Three regions out of four (Inner Hebrides, Outer Hebrides and Orkney) are estimated to be at carrying capacity, with the posterior mean carrying capacity parameter χ in each region (Table 1) very similar to the average observed number of pups in recent years (Fig. 3) and high precision of estimates relative to the prior distributions. The manner in which growth in pup production slowed over time in these regions was evidently informative about the form of the density dependence relationship, since the posterior on the density dependent shape parameter ρ was considerably more precise then the prior; the posterior mean of 5.6 was lower than the prior mean of 10 indicating that the steepness of the decline in fecundity as population size increases is less strong than envisaged a priori. In the North Sea region pup production is still rising at a near-constant rate and hence there is little information about the possible carrying capacity: the posterior distribution on χ_{NS} has a large overlap with the prior (Fig. 3).

Posterior correlations are jointly high between the two survival probability parameters ϕ_{pmax} and ϕ_a (Fig. D.3, Appendix D). This suggests weak identifiability of these parameters individually. For example, a simultaneous increase in adult survival and decrease in pup survival would have similar effects on pup production counts as an increase in pup survival with a decrease in adult survival. Due to this weak identifiability, it is difficult to distinguish which of these scenarios is most likely. Separate surveys with the aim of estimating a single one of these parameters (e.g., using mark-recapture) would be likely to increase the certainty in estimates of the other parameter as well, without necessitating other changes to the data collection process. Maximum juvenile survival ϕ_{pmax} is only observed when population size is close to zero, so it is not feasible to collect data on this parameter directly. Obtaining observations on $\phi_{t,r}$ may help identify other parameters—the observation model would need to be extended to include this, and a simulation study could be undertaken to determine the effect of adding information on this quantity.

The posterior distribution for the sex ratio parameter, ω is nearly identical to that of the prior. This is noted by Thomas et al. (2019), and is reflective of the fact that there is little information in the data on the number of adult males throughout the survey. Thomas et al. (2019) suggest that ω could be replaced by a male survival parameter if the number of unknown males in each region in each year were also considered latent states. The primary limitation to their implementation of this solution was computation time. Therefore, the accelerated method of inference we present in this article could offer an opportunity to explore this alternative, which allows for external information or expert elicitation on male survival to be included more readily.

We note the use of a normal distribution to model the (positive) pup production estimate in each region for each year. The use of a continuous distribution that places probability mass below zero is potentially problematic. Therefore, we utilise the greater number of effective samples from the posterior we have obtained to investigate this. Due to the current specification of the observation precision parameter ψ, the probability mass placed below zero is not dependent on the mean, but only on ψ. For the smallest value of ψ sampled from the posterior, a mass of only 4×10^{-18} is placed below zero. This is many orders of magnitude smaller than the machine epsilon (the smallest number that can be added to 1 without causing erroneous rounding back to the value of 1) in C, which is on the order of 1×10^{-7}. Similarly, the use of a continuity correction can make a difference of at most 3.5×10^{-7} in this case study. This difference is only possible for the largest value of ψ sampled by our MCMC chain. We are therefore confident that the use of a normal distribution in this modelling context is numerically equivalent to the use of a discrete distribution that places no probability mass below zero.

One advantage of the speed-up obtained by GPU parallelisation for our seal application and related nonlinear state-space models is that it becomes possible to use simulation studies to assess the accuracy of parameter and state estimation. This is relevant because issues have been noted with simpler linear normal state-space models, particularly in the case where the observation error is significantly larger than the process error (Auger-Méthé et al., 2016). We illustrate one such simulation in Appendix E. In that case we simulate 'ideal' data with multiple observations of pup and total population size as well as reducing the number of unknown parameters. We find posterior mean parameter estimates within 1% of the true value.

5.2. Case study: comparison with previous CPU-based analysis

Having used an APF method that involved kernel smoothing, Thomas et al. (2019) report an expected bias in samples from the posterior, although the amount of kernel smoothing used was very small and hence the size of bias expected to be small. One advantage of GPU parallelism in this modelling context is that it allows the use of an unbiased sampling method, via pMCMC. In general, the difference between the APF and pMCMC posterior marginal distributions were indeed small (Fig. 3). This suggests that bias introduced via kernel smoothing is minimal, and that it remains a valid method of inference for the case study at hand, computational efficiency aside.

We note a slight increase in the estimate of the observation error precision parameter ψ (i.e., higher estimated precision), and a decrease in the posterior standard deviation of this parameter, indicating a higher certainty on the observation error of pup production estimation. This provides more information to ecologists on the efficacy of data collection methods. Thomas et al. (2019) note that the introduction of a random effect on the fecundity parameter α is desirable. A reduction in the posterior standard deviation of ψ could reduce the extent to which it would be confounded with α if this random effect were introduced. We also note significant reductions in the posterior standard deviations of the density dependence parameter ρ, as well as all three of the carrying capacity parameters for which the population was no longer in the exponential growth phase (χ_{IH}, χ_{OH}, χ_{OR}, for the Inner Hebrides, Outer Hebrides, and Orkney survey areas, respectively). This allows more precise extrapolation to estimates of total population size or estimates of state values (the number of female seals in a given age group, in any particular year of the survey). Although the kernel smoothing procedure of the APF appeared to introduce little bias, the slight decoupling of states and parameters may have diluted the information in the data somewhat, and using an unbiased procedure, pMCMC, may therefore lead to more precise estimates on parameters that are well informed by the data.

5.3. Inference using GPUs: context and limitations

In this article we focused on a case study using particle filtering, whose potential for GPU-parallelism has been well explored (e.g. Henriksen et al., 2012; Lee et al., 2010b; Sipos et al., 2019). However, general tools that provide these implementations to users less familiar with GPU-specific programming practices (e.g. Murray, 2013) only provide support for a few standard modelling distributions, limiting their accessibility to a wider audience. Numerous other applications in ecology and evolution stand to gain from GPU parallelism also. Computationally expensive algorithms that involve performing independent tasks over many data points are prime candidates, e.g., those involving numerical integration (such as spatial capture recapture (Borchers and Efford, 2008; Efford, 2004; Royle and Young, 2008), where the unknown activity centre of individuals is marginalised, or Joint Species Distribution Modelling (JDSM. For an introduction, see Ovaskainen and Abrego, 2020), where marginalising occurs over latent variables that drive species co-occurrence). Similarly, algorithms that produce large quantities of random samples in parallel, such as Approximate Bayesian Computation, achieve impressive speedups when executed on GPUs (Kulkarni and Moritz, 2023). Bayesian inference algorithms often display potential for parallelism in this way, and the use of GPUs to leverage this is an active part of computational research in the field (e.g. Beam et al., 2014; Herbei and Berliner, 2014; Terenin et al., 2019; White and Porter, 2014; Xue et al., 2016).

Programs that contain large amounts of branching, such as simulations for Individual-Based Models (where the calculations required are often dependent on the state or life stage of an individual) are prone to 'warp divergence', which lowers the efficiency of GPUs (Chimeh and Richmond, 2018). Calculations with irregular memory access patterns, such as the Nested Sampling algorithm for Bayesian inference, are not readily suitable to GPU parallelism (Lewis et al., 2015). Further information on this point is available in Appendix B.

Algorithm suitability and design are not the only barriers to effective GPU-parallelism. As noted by Terenin et al. (2019), another significant hurdle is the lack of breadth within GPU-capable random number generation libraries. For example the cuRAND library for CUDA has only two options for continuous data: uniform and normal distributions. In many real-world modelling situations, observed data are truncated or come from asymmetrical distributions. These are hard to approximate using transformations of normal or uniform deviates and must often be sampled using rejection sampling, which displays warp divergence. In these cases, approximately 10^6 random numbers must be generated before GPUs become much faster than CPUs (Askar et al., 2021). The creation of open source libraries for GPU random number generation from a wide suite of standard distributions would facilitate the parallelising of many Monte Carlo algorithms.

GPU hardware is evolving, with the introduction of more specialised cores. One example is the Tensor Core, which can perform a fused multiplication and addition, allowing for more efficient matrix multiplications. However, consumer-grade cards continue to focus on calculations using single-precision floating point numbers (also called floats), with the two latest generations of Nvidia cards (Ampere and Ada Lovelace) providing only two double-precision cores per streaming multiprocessor, equating to a theoretical throughput 64 times smaller than that for single-precision floats (Krashinsky et al., 2020; NVIDIA Corporation, 2022). Therefore, the use of log-transformations and other algorithm adaptations prioritising high numerical stability will continue to form a core aspect of GPU programming for computational statistics applications. We note that in C and C++, the machine epsilon for single precision floats is approximately 1×10^{-7}. The lower the machine epsilon, the higher the precision of the calculation. Many high-precision applications, such as numerical integration for high-dimensional data, or p-value calculations within extreme value statistics may not be suitable for fine-grain parallelism without the creation of specialised hardware.

We benchmark results as an illustration of the potential for GPU code greatly reducing compute time, but we note this is not appropriate for use as a direct estimation of the differences between the best possible compute times achievable by CPUs and GPUs. This is because different architectures require different optimisation approaches (Lee et al., 2010a). In situations where threads of execution require large amounts of communication, or when sequential dependence occurs in nested **for** loops, existing CPU clusters are likely to remain a cost-effective method of reducing compute time, especially if implementations are ported over to compiled languages, such as C and Fortran. Tools such as Rcpp (Eddelbuettel, 2013), that allow the use of compiled code within a higher-level language are a vital first optimisation step for GPU-parallel applications as they provide a more-accessible and more widely documented method of code-acceleration.

5.4. Optimising aspects of the particle filter

The variance of an estimate of the likelihood derived by a particle filter is proportional to the inverse of the number of particles n_p (Moral, 2004). However, the efficiency of a particle filter's execution on a GPU is also dependent on n_p. Values of n_p that are too low will not utilise enough cores simultaneously to achieve high occupancy (a measure of the number of current parallel processes relative to a theoretical maximum). Values of n_p that are too high force the GPU to run calculations in serial batches, as there are too many particles for the number of cores available. This batching leads to memory and scheduling overheads that reduce overall performance. Therefore, it can be advantageous to split the total number of particles across different filters that are run in series. The mean of likelihood estimates obtained by these n_f filters is then used as the final estimate. We suggest performing a grid

search over small numbers of plausible values of n_f and n_p, as described in Section 3, to determine the combination that strikes the best balance between high occupancy, low compute time, and low variance for log-likelihood estimates. Situations where low values of n_p are selected along with high values of n_f lend themselves to a coarser grain of parallelism, making the use of CPU clusters more efficient. Similarly, it is easier to achieve higher occupancy on GPUs with fewer compute cores. This suggests that older hardware (which tends to contain fewer cores) may be considerably more cost effective for certain applications, as newer hardware will only achieve high occupancy if the number of parallel processes is sufficiently high. Therefore, newer hardware will be most efficient when fewer particle filters are used, each with a higher number of particles. GPU profiling tools are available to assess occupancy levels, so the ideal number of particle filters and total particles can be tuned to specific hardware. This is applicable more generally than just for particle filtering, as larger batches of calculations, each containing fewer processes, will achieve higher occupancy on older hardware. Cost efficiency for application-specific computations can therefore be obtained by profiling the occupancy of pilot computations on different GPUs in the cloud, so that hardware may be selected to achieve an optimal balance between price and computational throughput.

In many computational statistics algorithms, including particle filtering, resampling indices across a large set of data occurs regularly (such as in line 5 of Algorithm 1), and occupies a significant proportion of the total compute time. The choice of resampling algorithm is important, as algorithms that have efficient CPU-only implementations do not necessarily remain efficient when calculations are performed using GPUs. In particular, the rejection algorithm for multinomial resampling is prone to warp divergence, as threads in the same warp must wait for all others to have accepted a sample before they can proceed to generating the following sample). More examples of warp divergence are detailed in Appendix B. Efficient alternatives such as the Metropolis resampler (whose GPU-parallel performance is examined in detail by Murray et al. (2016)) or the Megopolis resampler (Chesser et al., 2022) are likely to reduce overall compute time by allowing each thread to produce a sample independently. However, as these algorithms are only asymptotically unbiased, they may not be optimal for applications that require resampling from highly skewed probability distributions, as these will necessitate many iterations to produce sufficiently unbiased samples, increasing compute time.

5.5. Conclusions

GPUs are becoming increasingly used within statistical computing, with both hardware and software now being well developed and readily accessible to scientists with little GPU-specific programming experience, for certain applications. In these applications, largely related to machine learning, approximately two orders of magnitude of speed-up can be obtained with relatively inexpensive consumer-grade equipment. For other custom applications such as state space modelling, knowledge of GPU-specific programming practices and lower-level programming languages are both still necessary to achieve comparable results. In this article we have outlined properties of statistical algorithms that make them well- or poorly-suited to GPU parallelism, and have provided examples in each case. Via a population dynamics case study, we have demonstrated how these concepts can be utilised to efficiently parallelise a computationally expensive algorithm for ecological inference, and we provide open source software for this case study that can be used as a template for similar applications.

Author contributions

Both authors conceived the project; C.F.-J. developed the algorithms and code, and led on drafting the paper, with input from L.T.

CRediT authorship contribution statement

Calliste Fagard-Jenkin: Conceptualization, Formal analysis, Methodology, Software, Writing – original draft, Writing – review & editing. **Len Thomas:** Conceptualization, Data curation, Formal analysis, Methodology, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare no conflicts of interest.

Data availability

Code and data required to replicate the main analysis are available on GitHub and linked to in the main text.

Acknowledgements

We thank all those responsible for collecting and processing the long-term data used in this article, in particular Callan Duck, Chris Morris and Debbie Russell of the Sea Mammal Research Unit. We thank Fanny Empacher for insightful discussions on model-fitting algorithms. We are grateful to the associate editor and an anonymous reviewers for their helpful comments and suggestions, which led to a substantially improved manuscript. C.F.-J. is funded via a doctoral scholarship from the University of St Andrews, School of Mathematics and Statistics.

Data and source code required to reproduce the case study in this article are available at https://github.com/calliste-fagard-jenkin/GPU-SSMs

Appendix A. Running CUDA code with google cloud

A.1. Introduction and specifications

This appendix is designed to provide practical advice on setting up a CUDA-capable Google cloud instance, as well as installing drivers and software required to use Nvidia GPUs for scientific computing. This file will cover both the execution of GPU-accelerated code using cloud computing, via the Google Cloud service, and the installation of Nvidia GPU drivers and compilers onto this server, if required. There are many other providers of similar services. However, they vary in their setup procedures and requirements, and so we provide only a detailed example for the above mentioned provider.

At the time of writing, CUDA-capable Google Cloud options use Linux operating systems (Debian 10, with CUDA 11.3), and the latest CUDA release is version 11.8. Given the lack of a simple way of converting a Windows executable to a Linux one when this contains GPU driver calls, the simplest way to run CUDA code in the cloud is to compile directly on the Linux server.

The set of instructions in this file will take as given a basic knowledge of CUDA code, familiarity with the C and C++ programming languages (namely makefiles, structs, templates, memory allocation and pointers. Familiarity with terminal windows in Linux or MacOS operating systems is also advantageous.

A.2. Installing Nvidia drivers and CUDA compilers

This first set of instructions details the installation of software required to use Nvidia GPUs for scientific computing. These instructions apply both to a local machine that contains CUDA-compatible GPUs, or to a server initialised using Google Cloud.

1. The machine should have a recent installation of the gcc C/C++ compiler
2. The `Nvidia CUDA Toolkit` and the Nvidia drivers for any GPUs in the machine should be installed. The CUDA compiler version can be verified with the terminal command `nvcc--version`. The driver versions and some basic diagnostics relating to the GPUs can be queried using the terminal command `nvidia-smi`.

A.3. Compiling and running code on a local machine

If a machine with a CUDA compatible GPU is available, with the above drivers installed, then scientific computing code can be compiled and run on this machine with the following instructions. See below for using GPUs in the cloud.

1. A GPU-accelerated project including a makefile, a main file, and any dependencies, can be compiled with the standard `make` command in a terminal window. Example projects, which can serve as a minimal reproducible example, can be found online. For an example of a more complicated project with a small handful of dependencies, see the source code made available with this article at https://github.com/calliste-fagard-jenkin/GPU-SSMs
2. In some instances, once the output file has been created by `make`, it may be necessary to formally add executable permissions to the file using the `chmod + X < filename>` command in a terminal window (open in the same file path as the executable file itself).

A.4. Using google cloud

1. A Google Cloud account should be accessible, or created if required. The account requires billing information for the use of GPUs. An application is required to increase the quota of available GPUs.
2. A CUDA-capable Virtual-Machine (VM) instance should be created on Google Cloud. Various choices are available by searching for 'CUDA' in the Google Cloud 'Marketplace'. We recommend a VM named "Debian 10 based Deep Learning VM with M97", as it comes with the necessary drivers and compilers pre-installed. At the time of writing, this instance uses CUDA 11.6. Documentation is available if additional information about similar Google Cloud VMs is required.
3. The VM can be launched using the 'Launch' button on the aforementioned page, after which the required hardware specification of the VM must be selected. For most small projects, 16GB of memory and an SSD boot drive of 30GB should suffice. The number and type of GPUs can be selected at this stage.
4. Once the VM is deployed, which can take a few minutes, the 'SSH' button on the VM's page should open a terminal window in the web-browser of the local computer, which will provide access to the server. There are other methods for accessing the server (directly via SSH for example. However, the browser terminal option is the simplest to use for individuals less familiar with the use of cloud computing).
5. When accessing the server, it is likely that the automatic download of some CUDA drivers on the VM will need to be confirmed.
6. At this point, a zip file of the GPU-code repository we wish to execute in the cloud can be transferred to the server. A gear icon in the top right corner of this browser terminal shell, or an upwards arrow icon, should open a window that allows the zip file's location on the local computer to be specified.
7. Once the file transfer is complete, the file can be unzipped on the server using the terminal command `unzip <filename.zip>`. Once the decompression is complete, it is necessary to navigate within this directory using the terminal `cd < target/directory/path>` command. From here, the terminal `make` command should compile the GPU-accelerated code. At this stage, it is possible that there is an incompatibility between the CUDA version specified in the makefile of the example code being used, and the CUDA version made available by the server. If this is the case, the CUDA version available to the server can be verified with the `nvidia-smi` command, and the makefile should be updated to reflect this.
8. Once the code is compiled, the executable file should become visible. As with code execution on the local machine, executable permissions may need to be given to this file using the `chmod +X < filename>` command. The executable can then be run on the server with the `./<executable-name>` command.
9. Once the executable has finished running, any output files can be transferred back to a local machine by clicking the gear icon in the top right of the browser terminal shell again (or a downwards arrow symbol), and selecting download. Once everything is transferred, it is important to stop the VM instance. If the instance will not need to be used in the near future, it is best practice to remove the instance entirely, to avoid the risk of accidental over-use, which will be charged.

Appendix B. Introduction to GPU computing concepts

B.1. Warps, blocks, and threads

This section aims to highlight the most important GPU programming concepts that greatly aid in understanding parallel architectures more generally, and also help to identify families of algorithms in computational statistics that could benefit from a GPU-accelerated implementation. For simplicity, Nvidia-specific terminology will be used to remain consistent with the main text.

Fig. 1c (main text) represents a GPU streaming multiprocessor (SM), which is a collection of cores with fast cache memory shared between them, and a small amount of resources for control structures. The longer block of memory in the middle of Fig. 1b (main text) is the GPU's global memory, and all information stored here is accessible by all cores (on the GPU), no matter which SM they belong to.

A single independent stream of computation is known as a thread. On traditional CPU-only machines, we can expect to be able to run up to 128

threads simultaneously, on high end consumer products. On GPUs thousands of threads can be run simultaneously with good management of resources, illustrating the level of parallelism we wish to leverage. Threads are organised into groups of 32 called 'warps'. All threads in a warp execute each of their instructions at the same time, known as running in 'lockstep'. These warps are organised into 'blocks', which can contain partial warps if required, with the unnecessary threads simply running idle. Finally, the set of all blocks is known as the 'grid'. The significance of these abstractions is that all threads in a given block will always be run on the same SM. This means they will all have access to the small cache displayed on the bottom of Fig. 1c (main text), allowing threads the opportunity to transfer information from one 'train of thought' to another. Threads in different blocks do not have this advantage. A final generality which strongly influences our programmatic decisions when working with GPUs is that the scheduling of blocks and warps cannot guarantee any specific sequence of operation: it is not possible to choose the particular order in which warps are run within a given block, or the order in which blocks will be executed.

In the remainder of this section, some general GPU programming concepts will be highlighted, to summarise the factors that typically have the largest impact on speed of calculation on a GPU. These are 'warp divergence', copying data between the host (the computer that houses the CPU and GPU) and the device (the GPU), coalesced memory access on the GPU, thread register use, and the significance of block and grid sizes (the number of threads per block, and the number of blocks requested by any GPU calculation, respectively).

B.2. Warp divergence

Warp divergence occurs as a result of the lockstep nature of threads undertaking their given tasks. Since all threads within a warp must execute each of their instructions simultaneously, threads that attempt to perform a different task to others in their warp will cause the remaining threads to idle as they wait. As an example consider the Box-Muller transform (Box, 1958) for generating normal deviates, given below as algorithm B.1.

Algorithm B.1. GPU-accelerated Box-Muller Algorithm for standard normal deviates

1: Set $i = \text{ThreadID} + \text{BlockID} * \text{ThreadsPerBlock}$.
2: **if** $i \geq N$ **then**
3: **return** NULL
4: **end if**
5: Sample $U_0, U_1 \sim \mathcal{U}(0, 1)$
6: Set $Z_0 = \sqrt{-2\ln(U_1)} \cdot \cos(2\pi U_2)$
7: Set $Z_1 = \sqrt{-2\ln(U_1)} \cdot \sin(2\pi U_2)$
8: Write Z_0 to the i^{th} index of the output vector
9: Write Z_1 to the $N + i^{\text{th}}$ index of the output vector
10: **return** NULL

Where $2N$ is the desired total number of standard normal deviates to produce.

Looking at Algorithm B.1 will help to illustrate some GPU-specific programming conventions, and then illustrate the type of algorithm that is ideally suited to GPU acceleration. The algorithm is written from the perspective of a single thread: to produce 2*N* standard-normal deviates, we write a function that instead focuses on a single thread of computation, which in this case produces only 2 of the required 2*N* deviates. This concept holds generally for GPU computing with CUDA, as 'kernels' (the name for a function that runs on the GPU) are always written for individual threads, allowing the parallelism to arise from launching this same kernel many hundreds of thousands, or millions of times. In this situation, each thread is aware of its unique identifier within its block of execution, and also of the unique identifier of this block within the collection of blocks which have been launched (known as the grid). By combining these unique identifiers with the number of threads contained in each block, a mapping can be produced to the set of indices $0, N-1$. This is performed by line 1 in the algorithm. Line 2 ensures that 'out of bounds' thread numbers do not result in unexpected behaviour. For example, consider producing 20 random deviates with 2 blocks containing 6 threads each. This setup would produce 12 threads, and values for *i* from 0 to 11. Since only 10 threads are required to produce 20 deviates, running this kernel for $i = 10$ or $i = 11$ may lead to attempts to access unassigned memory. The final convention to note is that many GPU-accelerated functions will often return NULL values. This is because there is no clean way to 'collect' the outputs of each thread and organise them neatly on our behalf. Each thread must know explicitly at which memory address(es) it is expected to write its outputs. Memory for this output array must be allocated before the kernel is executed.

Considering other lines of this algorithm, it is clear that for any instance of the kernel, and any value of *i*, calculations can be performed completely independently for each thread. The thread performing calculations for $i = 0$ requires no information produced by any other thread, and will be writing its output to indices 0 and *N* of the output array, which are not used by threads with any other value of *i*. This independence is key for optimal GPU-acceleration. If any information needed to be shared between threads, say on line 5 for example, then every other thread would need to be executed up until line 5 to guarantee that this information is available (since the order of execution is not guaranteed). We note that on line 3, threads with out of bounds values are asked to terminate immediately. In practice, this is not possible, and they must run through lines 4 to 10, waiting for all the other threads in their warp to reach the same line before moving on. These idle threads are wasted, since they perform no useful work, but must still be scheduled and run in order to obtain useful work from other threads in their warp. Because of this, choosing grid and block sizes that produce as few wasted threads as possible can have a significant effect on performance.

This 'idling' behaviour caused by warp divergence becomes dangerous when conditional statements in code lead to multiple useful threads performing different tasks. This is illustrated by considering an alternative version of the Box-Muller algorithm, based on using polar coordinates (Knop, 1969), shown in algorithm B.2. Lines 6 to 8 of the algorithm contain a **while** loop. Threads that have passed the check on line 2 must wait for all threads in their warp to keep performing this loop before they can move on, since all threads in the warp are in lockstep. In this example, the probability that this test fails is approximately 21.4%, and so we have an approximate 95% chance that an entire given warp waits for at least 10

iterations of the loop, potentially because of a single slow thread. In this case, the computational cost of this branch is low, which minimises the effect of the warp divergence. However, when branches are computationally expensive, even a rare occurrence of branching can have a drastic effect on performance. These situations are unfortunately extremely common in computational statistics, with rejection sampling being a simple example. Acceptance rates as high a 99% still lead to a 30% chance of a warp being slowed down (Thomas et al., 2009). GPU-accelerations are still beneficial if extreme care is taken with the choice of proposal distribution, so as to avoid high rejection rates. A proposal which is more computationally expensive to evaluate, but leads to a higher acceptance rate could produce higher throughput, despite its lower efficiency when used in a serial computation. This could lead to complicated and perhaps non-intuitive applied approaches (such as importance sampling within rejection sampling) becoming viable in certain situations.

Algorithm B.2. GPU-accelerated Polar Box-Muller Algorithm for standard-normal deviates

Set $i = \text{ThreadID} + \text{BlockID} * \text{ThreadsPerBlock}$.
if $i \geq N$ **then**
  return NULL
end if
Sample $U_0, U_1 \sim \mathcal{U}(0,1)$, and set $S = U_0^2 + U_1^2$
while $S \geq 1$ **do**
  Sample $U_0, U_1 \sim \mathcal{U}(0,1)$, and set $S = U_0^2 + U_1^2$
end while
Set $Z_0 = U_0 \cdot \sqrt{\frac{-2\ln(S)}{S}}$
Set $Z_1 = U_1 \cdot \sqrt{\frac{-2\ln(S)}{S}}$
Write Z_0 to the i^{th} index of the output vector
Write Z_1 to the $N + i^{\text{th}}$ index of the output vector
return NULL

Where $2N$ is the desired total number of standard normal deviates to produce.

Being aware of the possible sources of warp divergence, the proportion of threads we can expect to pursue the most expensive branches, and obtaining creative solutions to finding the balance between total throughput and speed of execution for a single thread is integral to efficient GPU implementations of scientific algorithms.

B.3. Memory transfer

One of the most common bottlenecks in GPU computing is the transfer of memory from the host to the device and vice versa. The performance of data transfer can be summarised by two properties of interest: latency and bandwidth. Latency refers to the length of time it takes to send any information between two points, and is often measured with a 'ping' time: the time it takes to transfer a given quantity of data from the starting point to the end point and back again. Bandwidth refers to the total volume of information that can be sent between two points in a given unit of time, typically measured on the order of GB/s in modern graphics cards. Generally, GPU memory bandwidth is higher than CPU memory bandwidth, with only high latency and low bandwidth memory transfers being available between the two (Fig. B.1).

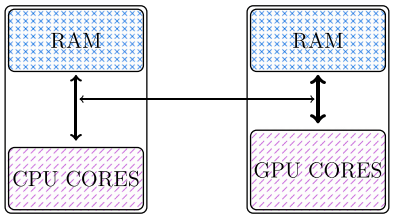


Fig. B.1. A diagram illustrating the transfer pathways between host memory, GPU memory, and compute cores. Longer arrows indicate higher latency, and thicker arrows indicate higher bandwidth. This figure is adapted from (Lee et al., 2010b, Fig. 2).

As the figure shows, GPU cores can communicate quickly with on-board GPU RAM to retrieve data mid-computation, but communicating with the CPU comes at a much greater cost. High latency means requesting or sending any data at all to the host comes at a relatively high cost, whilst the low bandwidth makes it infeasible to transfer large blocks of information to and from the CPU without threads sitting idle for many clock cycles. What the CPU lacks in volume, it gains in speed. Single threaded applications typically perform far better on the host, due to the CPU's higher clock speed per core, along with the presence of branch prediction and other optimisations provided by its control structures. However, copying data back to the CPU to perform 'inherently sequential' sections of an algorithm on the host is not necessarily optimal. In many situations, the additional data transfer time is greater than the time saved by using the CPU's faster clock-speed. Advanced profiling tools exist to determine which course of action is best for a given implementation, such as Nvidia's NSight (Nsight and Edition, 2013) for CUDA. When the burden of copying data from the GPU back to the host mid-algorithm is too great, writing GPU 'parallel' functions designed to be used by only a single thread is often the only solution. Sometimes a reduction can be used, allowing a simple task, such as the summation of values in an array, to be performed by many threads, each performing a single step of the calculation (in this case, adding together two 'adjacent' numbers). Repeating this process, halving the number of active threads with each iteration, until only a single thread is active, allows the vast majority of the summation to be performed in parallel. For detailed examples and

explanations on writing and profiling GPU reductions, see Sanders and Kandrot (2010).

A further important factor affecting performance is whether the GPU memory cache is accessed efficiently. The streaming multiprocessor running a block of threads provides them with a cache that has far lower latency than the GPU's global memory. Since threads are in lockstep, all requests to read memory as part of a given instruction are made simultaneously for the entire warp. If all threads in a warp make requests for information at consecutive memory addresses, which is known as a coalesced memory access, then all of these pieces of information can be loaded from the global memory using very few lines of cache. If the memory addresses requested by the threads are far apart, the line of memory sent to the cache to obtain one value will not contain the value required by another thread in the warp. This leads to many different lines of cache being read, causing a stall in code execution while memory latencies are resolved.

Consider Fig. B.2 and Fig. B.3, as well as algorithms B.3 and B.4, that illustrate how we can ensure coalesced memory accesses with our GPU programming. In the coalesced version (algorithm B.3), a perhaps non-intuitive increment is used: the product of the grid dimension and the block size. This obfuscates the number of operations n we wish each thread to perform, by making the number of passes through the **while** loop entirely dependent on the values of B (the block number of the active thread) and TPB (the number of threads in a given block). From Fig. B.2 we see this causes 'nearby' threads to access adjacent memory addresses at a given iteration of the loop, displayed by the grouping of arrow types. The non-coalesced version (algorithm B.4) has a more intuitive increment for each thread, but produces an inefficient memory access pattern. Whilst the same thread will access adjacent memory addresses at different iterations through the loop, during any given iteration, all of the 'nearby' threads are accessing memory addresses which are over a block-width away. Poor access patterns such as these can result in an order of magnitude of increased computation time when the quantity of information being read is substantial relative to the number of operations being performed (Sanders and Kandrot, 2010).

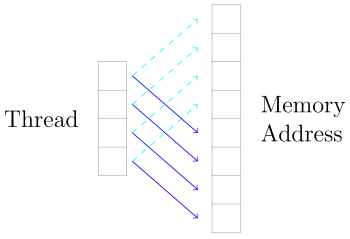


Fig. B.2. Example of coalesced memory accesses with each thread performing two tasks. Arrows with a dotted line show memory reads in the first time step, and arrows with a solid line show memory reads in the second time step.

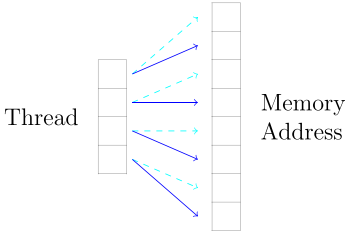


Fig. B.3. Example of non-coalesced memory accesses for each thread performing two tasks. Arrows with a dotted line show memory reads in the first time step, and arrows with a solid line show memory reads in the second time step.

Algorithm B.3. GPU number squaring with coalesced accesses.

1: Set $i = \mathrm{T} + \mathrm{B} * \mathrm{TPB}$
2: **while** $i < N$ **do**
3: output[i] = output[i]^2
4: $i \mathrel{+}= B * TPB$
5: **end while**
6: **return** NULL

Where T is the thread ID within the block, B the block ID within the grid, and TPB the threads present per block. N is the total number of numbers which we wish to square, and n is the quantity of numbers squared per thread.

Algorithm B.4. GPU number squaring without coalesced accesses.

```
Set i = n * T + n * B * TPB
for j in 0, ··· , n − 1 do
    if i + j < N then
        output[i + j] = output[i + j]^2
    end if
end for
return NULL
```

Where T is the thread ID within the block, B the block ID within the grid, and TPB the threads present per block. N is the total number of numbers which we wish to square, and n is the quantity of numbers squared per thread.

B.4. Other considerations

There are many other important factors that have substantial roles to play in efficient GPU programming that have been omitted. Most notably the existence of shared memory, which is accessible by all threads in a block, and is much faster to access than global memory. This can be useful when designing kernels in such a way that all threads in a block require access to the same parameter values.

Software abstractions relating to specific caching strategies also exist, and can have a strong effect on observed latency. One of these strategies made available by CUDA is 'texture memory'. Memory declared as texture memory is cached to prioritise spatial proximity as opposed to memory address proximity. As an example, the intensity of the colours red, green and blue in an image can be stored digitally in 3 matrices, each of same dimension as the image (in numbers of pixels). Texture memory will prioritise the caching of colour values for pixels that are close together in the image, even if the memory addresses where these values are stored are far apart. This can be beneficial when calculations for nearby pixels are dependent on the value of other pixels (such as when applying a Gaussian blur filter, for example). Typically this optimisation is used in image processing, but could be similarly advantageous in spatial statistics applications, or tasks in which a statistical function must be computed over a large grid space where parameter values are dependent on location.

Finally, the notion of 'thin' and 'fat' threads is useful, even if not explored here in detail. It suffices to understand that individual threads store a 'scratchpad' of variables they require in fast cache memory, using a software abstraction known as 'registers'. Because the cache these registers reside on is shared by all cores in a streaming multiprocessor, there is a strict upper limit on the number of concurrent warps an SM can schedule, dependent on the register space its threads require. Threads that require very little register space are known as 'thin', and lead to higher occupancy of the GPU hardware, whereas 'fat' threads require more register space and may be prone to having too few operations per memory read to hide the latencies these reads incur. Modifying the grid dimension and block size can optimise the execution time in these situations for a given graphics card. However, these types of optimisations rarely lead to improved performance across a wide range of hardware (especially across generations of graphics cards). In situations where threads are unavoidably 'fat', some variables or pieces of memory can be carefully re-purposed at different stages of the calculation to avoid as many distinct memory allocations as possible.

Appendix C. Benchmarking

In this section, we compare the efficacy of CPU-based parallelism and GPU based parallelism for state space model likelihood evaluation using particle filters. As the number of parallel cores used by any CPU-based implementation increases, the additional overheads this creates will place a practical ceiling on performance. The reduction in efficiency becomes extreme when the computational throughput required per compute core is low, relative to the latency of memory used to combine the results of calculations performed by different cores. In the case of the example application, this occurs when a small number of total particles is used.

We exemplify this with a small benchmarking simulation using two machines. The first machine was used to perform the analysis in the main text, and is equipped with two RTX 3070 GPUs (5888 'CUDA cores' per GPU). The second was equipped with an Intel i9-9880H CPU (8 cores) and a Quadro 4000 laptop GPU (2304 'CUDA cores'). The execution times of calculations performed using a single core of the latop computer were used as a baseline (a speedup factor of 1).

It is important to note that overheads may skew comparisons when comparing compute times. Therefore, for the CPU-based implementation, the time taken to create a parallel cluster was excluded. In the case of the GPU-based implementation, the time taken to allocate GPU memory, and to generate initial seeds for the pseudo-random number generators was excluded. This is justified by the tens of thousands of samples typically produced over the course of pMCMC (or tens of thousands of particles, during APF model-fitting), that eclipse the compute time of the initial overheads, which only need to be performed once.

Referring to Table C.1, we note that when the number of parallel processes is too low, GPUs provide less relative speedup (the ratio of the speedup factor for GPUs relative to 8 Laptop CPUs is lowest for the smallest number of total particles we considered). We also note that as the number of cores in a GPU increases, so does the number of parallel processes required for efficiency (the number of total particles at which the maximum speedup is achieved is higher for the desktop GPU than for the laptop GPU, and higher still when using two desktop GPUs). Using an additional GPU results in a 50–70% increase to the speedup factor, when computational throughput is sufficiently high. The Quadro 4000 mobile card demonstrates that when using laptop computers, GPUs are capable of accelerating computations a further 20 to 30 fold relative to using eight CPU cores, and by approximately

two orders of magnitude compared to using a single CPU core.

Table C.1
Speedup ratios of particle filter evaluations, compared to calculations performed with a single CPU core, estimated by timing the execution of 50 particle filters for the log-likelihood of the grey seal model, described in the main text. All particle filters were evaluated at the *maximum* a posteriori (MAP) of the analysis in the main text.

Total particles	8 Laptop CPU Cores	Laptop GPU	Desktop GPU	Two Desktop GPUs
2^{10}	1.87	13.16	16.67	15.93
2^{15}	4.37	117.27	167.16	262.64
2^{16}	4.18	106.24	174.43	265.37
2^{17}	4.98	97.36	166.43	279.21
2^{18}	4.90	80.36	137.59	236.24

Appendix D. MCMC diagnostics

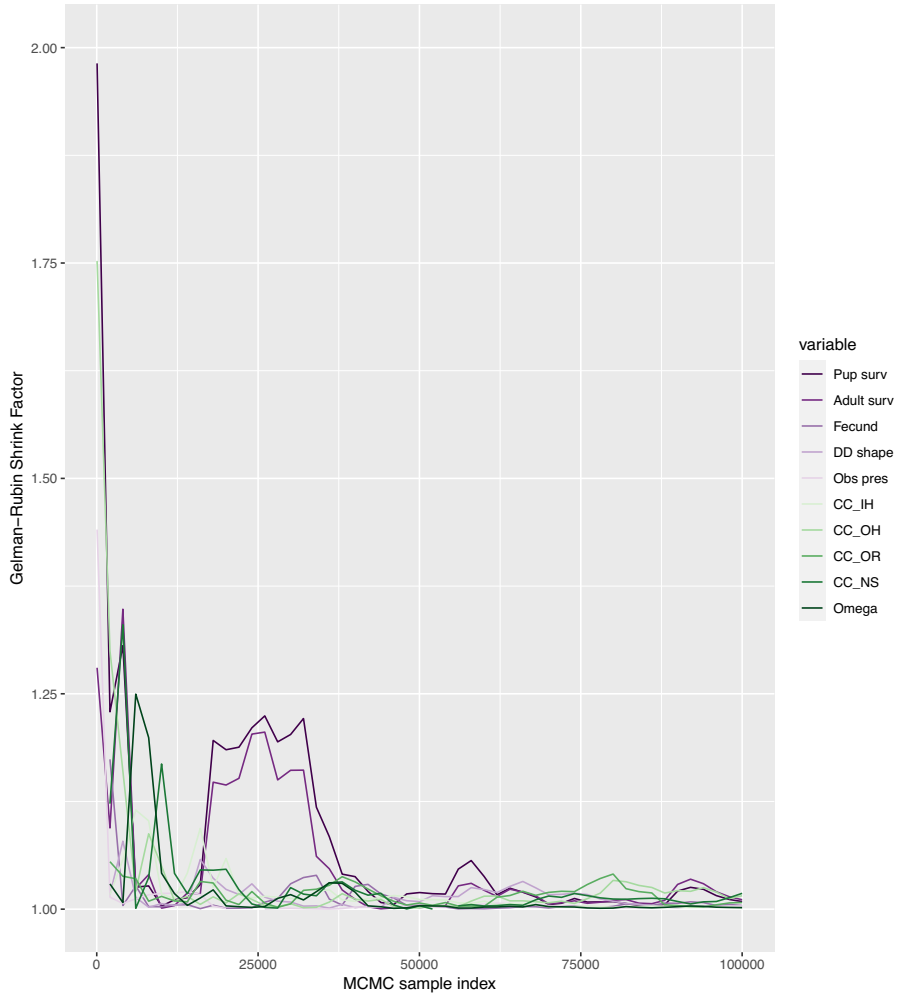


Fig. D.1. Grey seal case study median Gelman-Rubin shrink factor for all model parameters. Calculated using all 10^5 samples from both chains, with an initial bin of 50 samples.

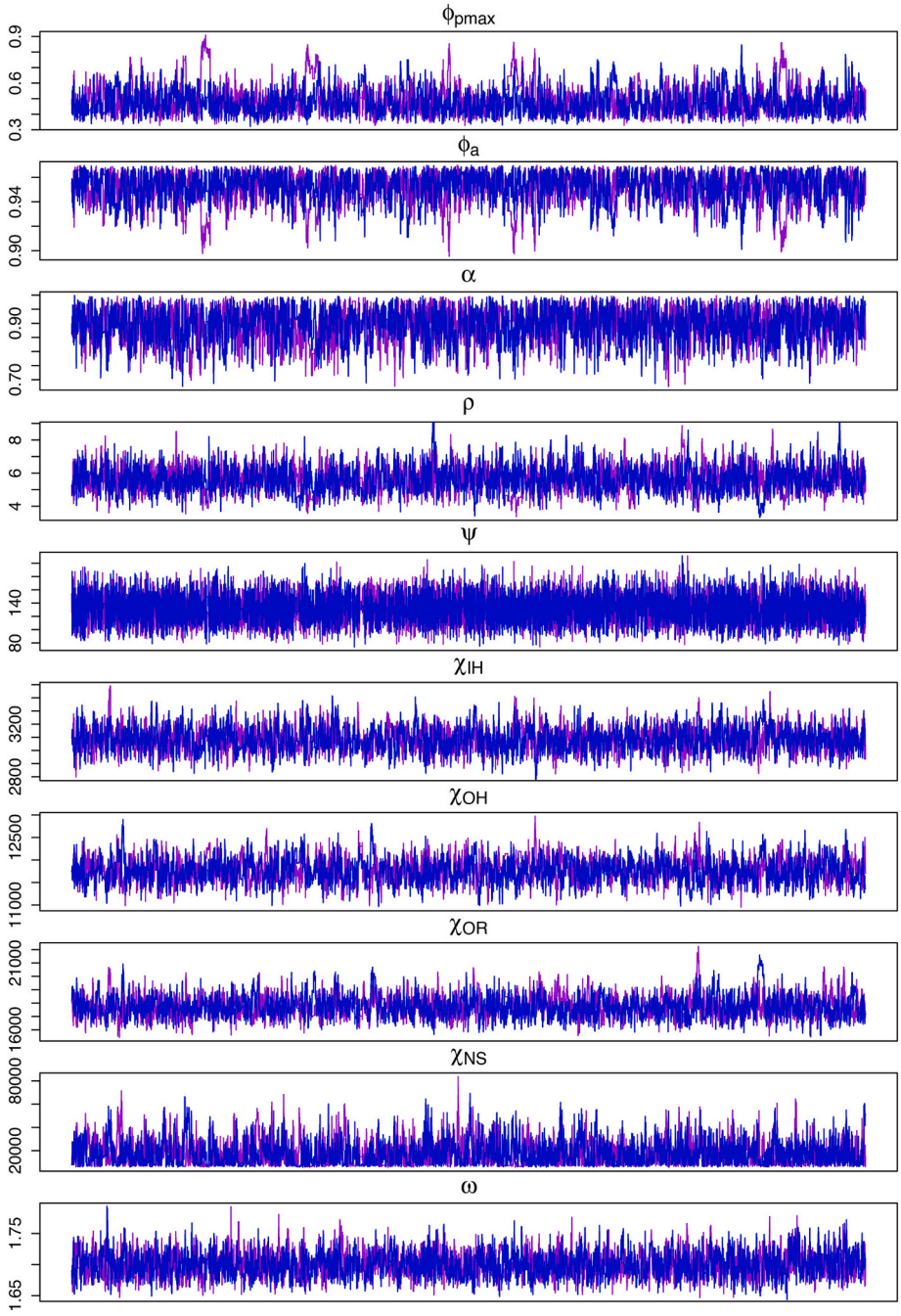


Fig. D.2. Grey seal case study MCMC traces. No samples have been removed via thinning. Each colour represents a distinct MCMC chain. Each trace displays all 10^5 samples from the posterior.

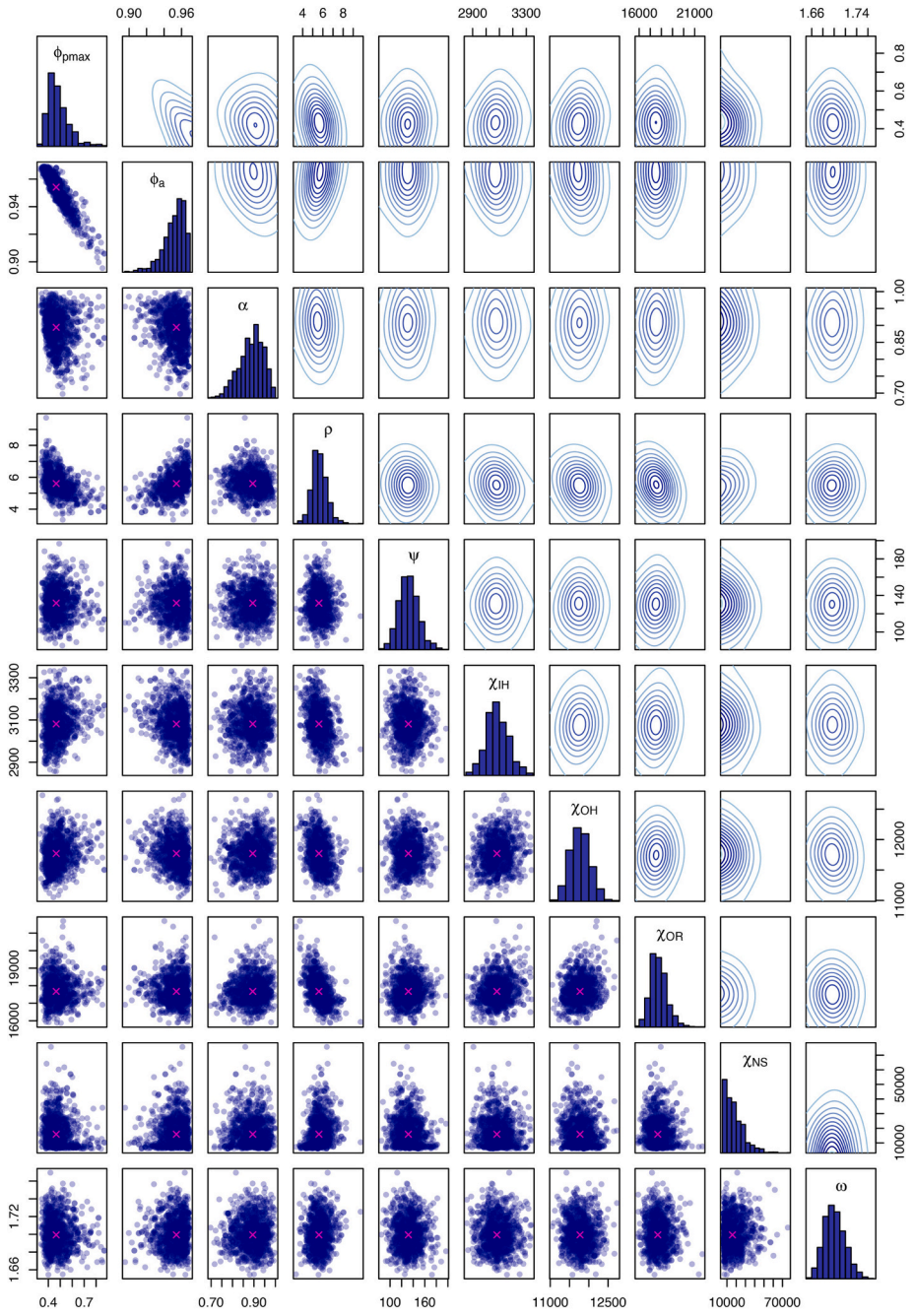


Fig. D.3. Grey seal case study two-way marginal distribution plots. Boxes on the diagonal contain histograms of marginal posterior distributions for each parameter. Boxes in the upper-right triangle contain plots from a kernel density estimate of two-way marginal distributions. Boxes in the lower-left triangle contain the two-way locations of 1000 samples from the posterior, with means indicated by a cross.

Appendix E. Simulation from 'ideal' conditions

We illustrate the effectiveness of the GPU pMCMC method for parameter inference via a simulation of the wildlife population dynamics model for grey seals. We aim to show that given 'high quality' data, the method is capable of producing posterior distributions whose marginal means are extremely close to truth. In this context, we define truth as the parameter values from which simulated data were produced.

When the observation error is larger than the process error, state space models are prone to parameter estimation issues (Auger-Méthé et al., 2016). In the grey seal case study presented in this article, pup production estimates are made with an observation error with a coefficient of variation (CV) of approximately 9% ($\phi^{-0.5}$, substituting ϕ for the posterior mean for this parameter, obtained from the analysis performed in the main text). However, the process that generates pup counts has a CV of $\sqrt{\frac{1-\alpha}{n\alpha}}$ (as it arises from a binomial distribution), where n is the number of breeding age females in a given region, in a given year. For plausible values of n, this quickly falls below 1%. This high observation error, combined with high posterior correlations between ϕ_{pmax}, ϕ_a, and α, can cause the means of their marginal posterior distributions to deviate from truth. To remedy this issue for the purposes of our illustration, we simplify the model by treating five of the model's parameters as known values. A more detailed investigation is planned

for the future.

We simulated new pup production estimates for all regions, extending the survey to a 100 year time series across the four regions, with an independent estimate of total adult population size occurring each year. Initial pup production counts in year 0 were fixed at the values given by Thomas et al. (2019). Marginal posterior correlations between parameters (especially between the vital rates) were minimised by fixing all model parameters other than ϕ_a and the four carrying capacities $\chi_{IH}, \chi_{OH}, \chi_{OR}$, and χ_{NS}. To ensure that biologically plausible parameter values were chosen to conduct the simulation, the posterior centroid of an initial analysis on the Thomas et al. (2019) data was selected. We fixed $\psi = 816$, to ensure a CV of pup production estimates of 3.5%. Similarly, we fixed $k_1 = 10^4$ and $k_2 = 3.523067$, to obtain independent estimates of total population size with CV 1%.

Two MCMC chains of 5×10^4 were produced. The proposal covariance structure was obtained via the same iterative process described in the main text. Samples from these iterative chains were discarded, and therefore no samples from the final chains were removed as a 'burn in'. Total compute time was approximately 6.2 h once the final covariance structure was obtained (and approximately 10.3 h including the adaptive phase). Post processing time was negligible. We then compared posterior estimates of parameter values with the known values observations were simulated from. We also assessed the method's ability to produce accurate inference on the values of hidden states, by simulating population trajectories from both the true parameter values, and parameter values sampled from the posterior distribution.

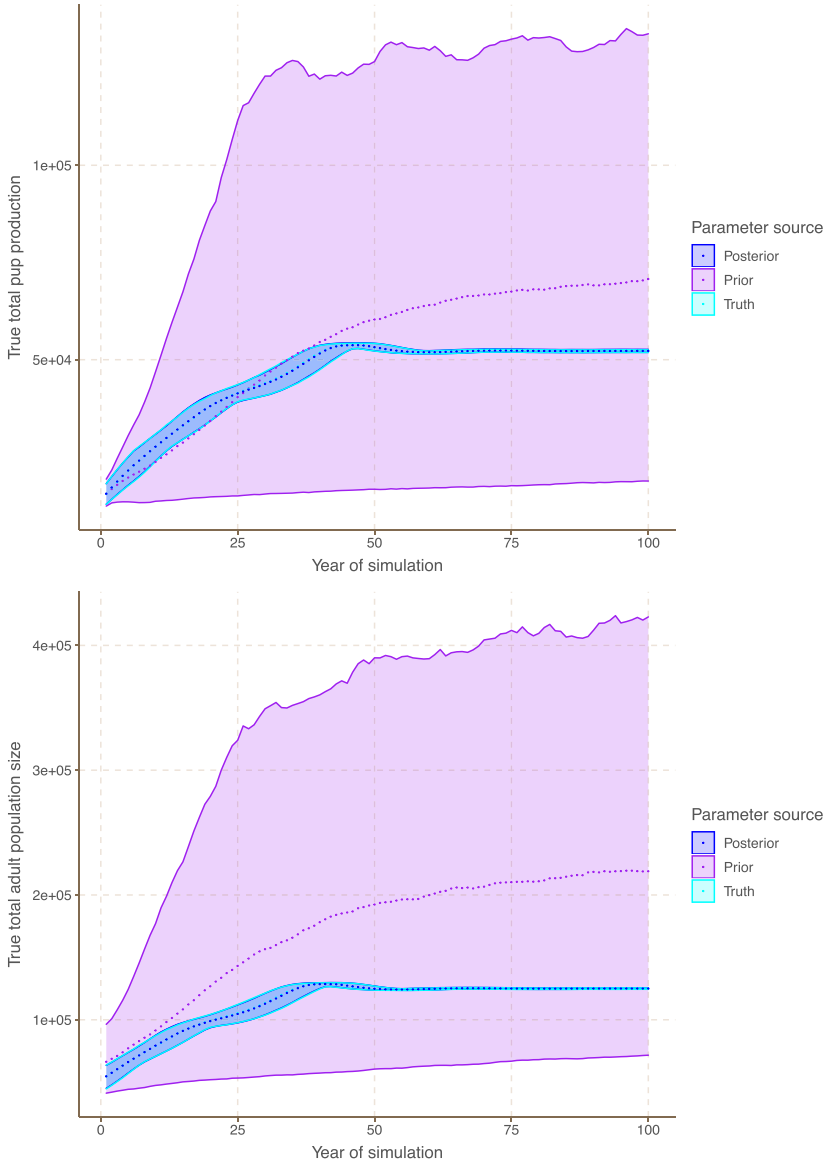


Fig. E.1. Simulation of 5000 grey seal time series, with model parameters coming from the posterior, prior, and truth (values used to simulate the data used to produce the posterior). The median of total pup production counts and total adult counts, across the four survey regions, is plotted for each of the 100 simulated years. The 2.5% and 97.5% quantiles for each count are plotted as solid lines. The prior is diffuse, and the posterior and truth curves are similar in each year.

From Table E.1, we show that the SSM framework (implemented via GPU pMCMC) is capable of producing posterior means that are within 1% of the true parameter value, when the observation error is sufficiently low. When simulating new populations, taking parameter values from the posterior distribution, or repeatedly using the same parameter values as those used to obtain the posterior, we obtain pup production counts and total adult counts with similar means, 2.5%, and 97.5% quantiles (Fig. E.1).

Table E.1
True and posterior mean parameter values, and their percentage differences, to 3 significant figures.

	Truth	Estimate	Difference (%)
ϕ_a	0.95614	0.95618	0.004
χ_{IH}	3083.8	3091.0	0.235
χ_{OH}	11,752	11,822	0.599
χ_{OR}	17,780	17,665	0.645
χ_{NS}	19,588	19,671	0.421

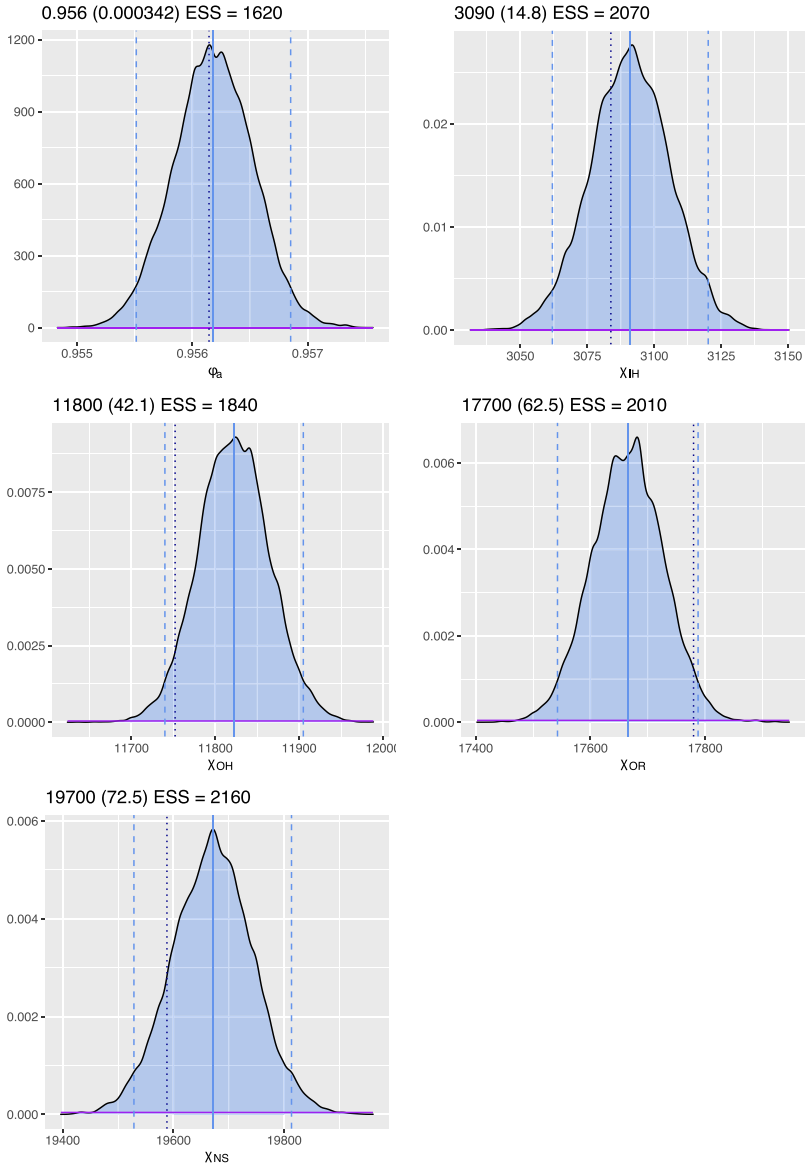


Fig. E.2. Marginal posterior distributions for each estimated parameter. Individual plot titles provide the marginal posterior means, standard deviations (in parentheses) and effective sample sizes. Posterior means are indicated by a solid vertical line. 95% credible intervals are indicated by dashed vertical lines. The parameter value the data were produced from is indicated by a vertical dotted line. Posterior distributions are indicated by a curve with a solid line (with shading below the curve). Prior distributions are indicated by a curve with a solid line and no shading below the curve (note these have very little density, relative to the posterior). Prior means are not displayed, to enable the shape of posterior marginal distributions to remain clear.

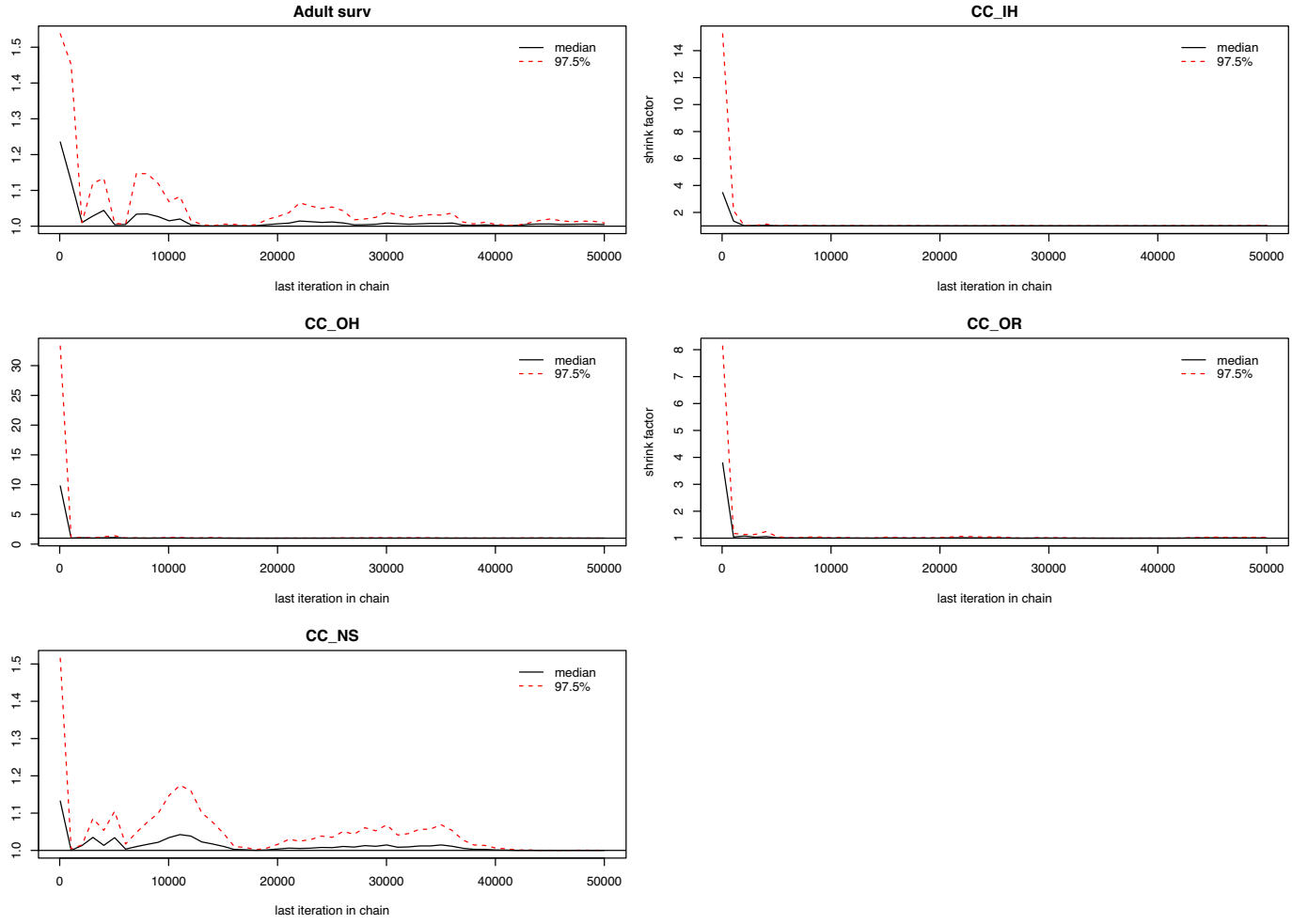


Fig. E.3. Gelman-Rubin convergence diagnostic for the estimated parameters.

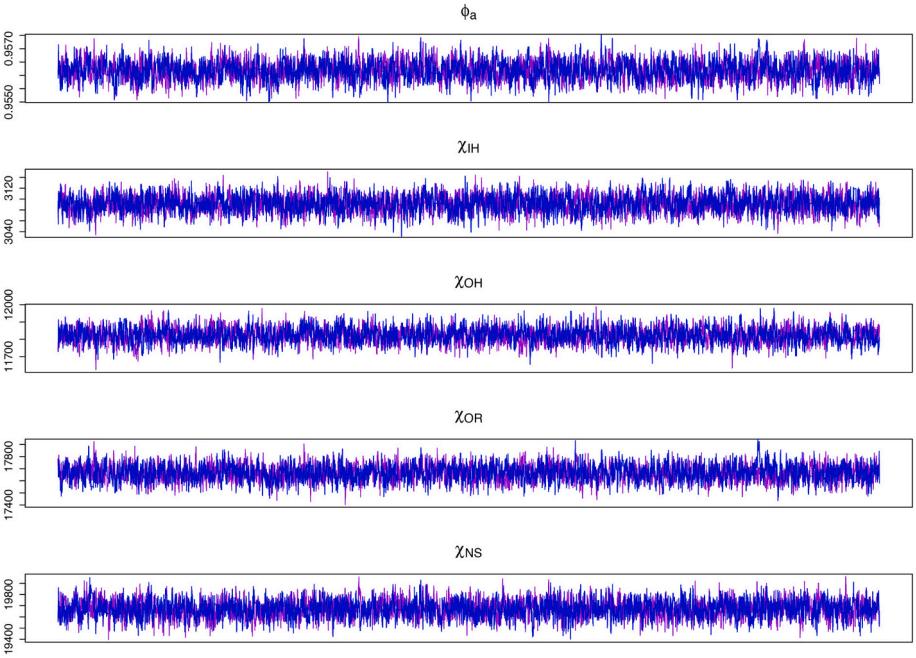


Fig. E.4. Final traces for the estimated parameters.

Appendix F. pMCMC algorithm

Algorithm F.1. Bootstrap Particle Filter for Marginal Log Likelihood Estimation.

set $LL := 0$
for each particle, i **do** sample $\boldsymbol{x}_0^i \sim \zeta$ from the prior **end for**
for each particle, i **do** set the log weight $\boldsymbol{w}_i := 0$ **end for**
for $t \in 1{:}T$ **do**
  for each particle, i **do** resample an index $I_i \sim \propto \exp w_{1:N}$ **end for**
  for each particle, i **do** reset $w_i := 0$ **end for**
  for each particle, i **do**
    set $\tilde{x}^i_{t-1,r,1:A} := x^{I_i}_{t-1,r,1:A}$
    generate $x^i_{t,r,1:A} \sim g(x^i_t|\tilde{x}^i_{t-1}, \theta)$
    $w_i \mathrel{+}= \log f(y^i_t|x^i_t, \theta)$
  end for
  $LL \mathrel{+}= \log \sum_{i=1}^{N} (\exp\{w_i - M\}) - \log N + M$
end for
return LL;

There are observations at T evenly spaced time points. N simulated particles are used to estimate the log likelihood LL. The function ζ is the prior distribution for the initial states. The function $g(x_t|\tilde{x}_{t-1}, \theta)$ is known as the state transition density. Simulations from this process are used to project particles to the following time step. $f(y_t|x_t)$ is the observation density, and describes the probability density of our observations given the hidden state values for a given time point. M is any number approximately the size of the largest log weight, and helps to ensure numerical stability. Lines of pseudo-code within a **for each** loop are not dependent on the result of the same calculation for previous indices, and therefore have the potential to be performed in parallel.

Algorithm F.2. pMCMC via the Metropolis-Hastings Algorithm.

set θ, the vector of starting model parameter values.
set Σ, the variance-covariance matrix of the random walk proposal distribution.
set $LL = \log(f(x|\theta))$, using the bootstrap particle filter
set $LP = \log(\pi(\theta)) + LL$
for $i \in 1{:}N$ **do**
  generate $\theta^* \sim N(\theta, \Sigma)$
  generate $U \sim U(0, 1)$
  calculate $LL^* = \log(f(x|\theta^*))$, using the bootstrap particle filter
  calculate $LP^* = \log(\pi(\theta^*)) + LL$
  if $U < \exp(LP - LP*)$ **then**
    $LL = LL*; LP = LP*; \theta = \theta*$
  end if
  record $\theta_i = \theta$, the i^{th} sample from the MCMC chain
end for

N denotes the pre-determined number of samples to be obtained from the MCMC chain. Σ and the initial value of θ are tuning parameters that will affect the efficiency of the resulting MCMC chain. $\pi(\theta)$ denotes the prior density of the model parameters θ. The log-likelihood, LL, is the quantity estimated by the bootstrap particle filter (Algorithm 1). $N(\theta, \Sigma)$ denotes a multivariate normal distribution with mean θ and variance-covariance matrix Σ. $U(0, 1)$ denotes a uniform distribution on the range $(0, 1)$.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X., 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org. URL:. *https://www.tensorflow.org/*.

Alsmirat, M.A., Jararweh, Y., Al-Ayyoub, M., Shehab, M.A., Gupta, B.B., 2017. Accelerating compute intensive medical imaging segmentation algorithms using hybrid CPU-GPU implementations. Multimed. Tools Appl. 76, 3537–3555.

Anderson, J.A., Lorenz, C.D., Travesset, A., 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Comput. Phys. 227 (10), 5342–5359.
Andrieu, C., Roberts, G.O., 2009. The pseudo-marginal approach for efficient Monte Carlo computations. Ann. Stat. 37 (2), 697–725.
Askar, T., Shukirgaliyev, B., Lukac, M., Abdikamalov, E., 2021. Evaluation of pseudo-random number generation on GPU cards. Computation 9 (12), 142.
Atila, Ü., Uçar, M., Akyol, K., Uçar, E., 2021. Plant leaf disease classification using EfficientNet deep learning model. Eco. Inform. 61, 101182.
Auger-Méthé, M., Field, C., Albertsen, C.M., Derocher, A.E., Lewis, M.A., Jonsen, I.D., Flemming, J.M., 2016. State-space models' dirty little secrets: even simple linear Gaussian models can have estimation problems. Sci. Rep. 6 (1), 1–10.
Auger-Méthé, M., Newman, K., Cole, D., Empacher, F., Gryba, R., King, A.A., Leos-Barajas, V., Mills Flemming, J., Nielsen, A., Petris, G., Thomas, L., 2021. A guide to state–space modeling of ecological time series. Ecol. Monogr. 91 (4), e01470.
Beale, M.H., Hagan, M.T., Demuth, H.B., 2018. Deep Learning Toolbox. User's Guide. The MathWorks, Inc., Natick, MA.
Beam, A., Ghosh, S., Doyle, J., 2014. Fast Hamiltonian Monte Carlo using GPU computing. arXiv preprint, p. **1035724**.
Bisset, K.R., Aji, A.M., Marathe, M.V., Feng, W.-C., 2012. High-performance biocomputing for simulating the spread of contagion over large contact networks. BMC Genomics 13 (2), 1–12.
Borchers, D.L., Efford, M., 2008. Spatially explicit maximum likelihood methods for capture–recapture studies. Biometrics 64 (2), 377–385.
Borowska, A., King, R., 2022. Semi-complete data augmentation for efficient state space model fitting. J. Comput. Graph. Stat. 1–40.
Box, G.E., 1958. A note on the generation of random normal deviates. Ann. Math. Stat. 29, 610–611.
Brooks, S., Gelman, A., Jones, G., Meng, X.-L., 2011. Handbook of Markov Chain Monte Carlo. CRC Press.
Buckland, S., Newman, K., Thomas, L., Koesters, N., 2004. State-space models for the dynamics of wild animal populations. Ecol. Model. 171 (1–2), 157–175.
Chai, J., Su, H., Wen, M., Cai, X., Wu, N., Zhang, C., 2013. Resource-efficient utilization of CPU/GPU-based heterogeneous supercomputers for Bayesian phylogenetic inference. J. Supercomput. 66 (1), 364–380.
Chang, W., Luraschi, J., Mastny, T., 2020. profvis: interactive visualizations for profiling R code. R package version 0.3.7. **URL:**. *https://CRAN.R-project.org/package=profvis*.
Chesser, J.A., Van Nguyen, H., Ranasinghe, D.C., 2022. The Megopolis resampler: memory coalesced resampling on GPUs. Digit. Signal Process. 120, 103261.
Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E., 2014. cudnn: efficient primitives for deep learning', *arXiv preprint arXiv:1410.0759*.
Chimeh, M.K., Richmond, P., 2018. Simulating heterogeneous behaviours in complex systems on GPUs. Simul. Model. Pract. Theory 83, 3–17.
Chollet, F., et al., 2015. Keras. `https://keras.io`.
Chopp, D.L., 2019. Introduction to High Performance Scientific Computing, vol. 30. SIAM.
Cook, S., 2012. CUDA Programming: A developer's Guide to Parallel Computing with GPUs. Newnes.
Del Moral, P., 1996. Non-linear filtering: interacting particle resolution. Markov Processes Related Fields 2 (4), 555–581.
Doucet, A., Pitt, M.K., Deligiannidis, G., Kohn, R., 2015. Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator. Biometrika 102 (2), 295–313.
Eddelbuettel, D., 2013. Seamless R and C++ Integration with Rcpp. Springer, New York.
Efford, M., 2004. Density estimation in live-trapping studies. Oikos 106 (3), 598–610.
Eklund, A., Dufort, P., Villani, M., LaConte, S., 2014. BROCCOLI: software for fast fMRI analysis on many-core CPUs and GPUs. Front. Neuroinform. 8, 24.
Endo, A., van Leeuwen, E., Baguelin, M., 2019. Introduction to particle Markov-chain Monte Carlo for disease dynamics modellers. Epidemics 29, 100363.
Farber, R., 2011. CUDA Application Design and Development. Elsevier.
Filho, A.R., Martins de Paula, L.C., Coelho, C.J., de Lima, T.W., da Silva Soares, A., 2016. CUDA parallel programming for simulation of epidemiological models based on individuals. Math. Methods Appl. Sci. 39 (3), 405–411.
Finke, A., King, R., Beskos, A., Dellaportas, P., 2019. Efficient sequential Monte Carlo algorithms for integrated population models. J. Agric. Biol. Environ. Stat. 24 (2), 204–224.
García-Feal, O., González-Cao, J., Gómez-Gesteira, M., Cea, L., Domínguez, J.M., Formella, A., 2018. An accelerated tool for flood modelling based on Iber. Water 10 (10), 1459.
Gelman, A., Carlin, J.B., Stern, H.S., Rubin, D.B., 1995. Bayesian data analysis. Chapman and Hall/CRC.
Gordon, N.J., Salmond, D.J., Smith, A.F., 1993. Novel approach to nonlinear non-Gaussian Bayesian state estimation. IEE Proceedings F (Radar and Signal Processing) 140, 107–113.
Guo, D., Shi, W., Qian, F., Wang, S., Cai, C., 2022. Monitoring the spatiotemporal change of Dongting Lake wetland by integrating Landsat and MODIS images, from 2001 to 2020. Eco. Inform. 72, 101848.
Henriksen, S., Wills, A., Schön, T.B., Ninness, B., 2012. Parallel implementation of particle MCMC methods on a GPU. IFAC Proceedings Volumes 45 (16), 1143–1148.
Herbei, R., Berliner, L.M., 2014. Estimating ocean circulation: an MCMC approach with approximated likelihoods via the Bernoulli factory. J. Am. Stat. Assoc. 109 (507), 944–954.
Hou, Q., Miao, C., Chen, S., Sun, Z., Karemat, A., 2023. A Lagrangian particle model on GPU for contaminant transport in groundwater. Comput. Particle Mechan. 10 (3), 587–601.
Kalman, R.E., 1960. A new approach to linear filtering and prediction problems. Trans. ASME–J. Basic Eng. 82 (Series D), 35–45.
Kamewar, A.S., 2017. Processing geospatial images using GPU. In: 2017 International Conference on Emerging Trends & Innovation in ICT (ICEI). IEEE, pp. 27–32.
Kattwinkel, M., Reichert, P., 2017. Bayesian parameter inference for individual-based models using a Particle Markov Chain Monte Carlo method. Environ. Model Softw. 87, 110–119.
King, R., 2011. Statistical ecology. In: Brooks, S., Gelman, A., Jones, G., Meng, X.-L. (Eds.), Handbook of Markov Chain Monte Carlo: Methods and Applications. CRC Press, pp. 419–447.
Knape, J., 2008. Estimability of density dependence in models of time series data. Ecology 89 (11), 2994–3000.
Knape, J., De Valpine, P., 2012. Fitting complex population models by combining particle filters with Markov chain Monte Carlo. Ecology 93 (2), 256–263.
Knobloch, M., Mohr, B., 2020. Tools for GPU computing–debugging and performance analysis of heterogenous HPC applications. Supercomput. Front. Innov. 7 (1), 91–111.
Knop, R., 1969. Remark on algorithm 334 [G5]: normal random deviates. Commun. ACM 12 (5), 281.
Krashinsky, R., Giroux, O., Jones, S., Stam, N., Ramaswamy, S., 2020. NVIDIA Ampere Architecture in-Depth. NVIDIA Developer Blog.
Kulkarni, S., Moritz, C.A., 2023. Improving effectiveness of simulation-based inference in the massively parallel regime. IEEE Trans. Parallel Distrib. Syst. 34 (4), 1100–1114.
Lee, A., Whiteley, N., 2016. Forest resampling for distributed sequential Monte Carlo. Statistical Analysis Data Mining ASA Data Sci. J. 9 (4), 230–248.
Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., 2010a. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proceedings of the 37th annual international symposium on Computer architecture, pp. 451–460.
Lee, A., Yau, C., Giles, M.B., Doucet, A., Holmes, C.C., 2010b. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. J. Comput. Graph. Stat. 19 (4), 769–789.
Leonenko, V.N., Pertsev, N.V., Artzrouni, M., 2015. Using high performance algorithms for the hybrid simulation of disease dynamics on CPU and GPU. Proc. Computer Sci. 51, 150–159.
Lewis, S., Ireland, D., Vanderbauwhede, W., 2015. Code optimisation in a nested-sampling algorithm. Nucl. Instrum. Methods Phys. Res., Sect. A 785, 105–109.
Link, W.A., Eaton, M.J., 2012. On thinning of chains in MCMC. Methods Ecol. Evol. 3 (1), 112–115.
Liu, J., West, M., 2001. Combined parameter and state estimation in simulation-based filtering. In: Sequential Monte Carlo methods in Practice. Springer, pp. 197–223.
Millán, E.N., Goirán, S.B., Piccoli, M.F., García Garino, C., Aranibar, J.N., Bringa, E.M., 2016. Monte Carlo simulations of settlement dynamics in GPUs. Clust. Comput. 19 (1), 557–566.
Moral, P., 2004. Feynman-Kac Formulae: Genealogical and Interacting Particle Systems with Applications. Springer.
Mueller, C.L., 2010. Exploring the common concepts of adaptive MCMC and covariance matrix adaptation schemes. In: Auger, A., Shapiro, J.L., Whitley, L.D., Witt, C. (Eds.), 'Theory of Evolutionary Algorithms', Vol. 10361 of *Dagstuhl Seminar Proceedings (DagSemProc)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 1–10. URL. https://drops.dagstuhl.de/opus/volltexte/2010/2813.
Murray, L.M., 2013. Bayesian state-space modelling on high-performance hardware using LibBi arXiv preprint arXiv:1306.3277.
Murray, L.M., Lee, A., Jacob, P.E., 2016. Parallel resampling in the particle filter. J. Comput. Graph. Stat. 25 (3), 789–805.
Nanni, L., Maguolo, G., Paci, M., 2020. Data augmentation approaches for improving animal audio classification. Eco. Inform. 57, 101084.
Newman, K., Buckland, S., Morgan, B.J., King, R., Borchers, D., Cole, D.J., Besbeas, P., Gimenez, O., Thomas, L., 2014. 'Modelling Population Dynamics', *Modelling Population Dynamics: Model Formulation, Fitting and Assessment Using State-Space Methods*. Springer, New York, New York, USA, pp. 169–195.
Newman, K., King, R., Elvira, V., de Valpine, P., McCrea, R.S., Morgan, B.J., 2023. State-space models for ecological time-series data: practical model-fitting. Methods Ecol. Evol. 14 (1), 26–42.
Nsight, N., Edition, V.S., 2013. 3.0 User Guide. NVIDIA Corporation.
NVIDIA Corporation, 2022. NVIDIA Ada GPU Architecture. https://images.nvidia.com/aem-dam/Solutions/geforce/ada/ada-lovelace-architecture/nvidia-ada-GPU-architecture-whitepaper-1.03.pdf. Accessed: 2023-01-19.
NVIDIA, Vingelmann, P., Fitzek, F.H., 2020. CUDA, release: 10.2.89. URL:. *https://developer.nvidia.com/cuda-toolkit*.
Ovaskainen, O., Abrego, N., 2020. Joint Species Distribution Modelling: With Applications in R. Cambridge University Press.
Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. Pytorch: an imperative style, high-performance deep learning library. Adv. Neural Inf. Proces. Syst. 32.
Perez-Wohlfeil, E., Trelles, O., Guil, N., 2023. Irregular alignment of arbitrarily long DNA sequences on GPU. J. Supercomput. 79 (8), 8699–8728.
Peters, G.W., Hosack, G.R., Hayes, K.R., 2010. 'Ecological non-linear state space model selection via adaptive particle Markov chain Monte Carlo (AdPMCMC)', *arXiv preprint arXiv:1005.2238*.
Pichler, M., Hartig, F., 2022. 'Machine Learning and Deep Learning–A review for Ecologists', *arXiv preprint arXiv:2204.05023*.

Pitt, M.K., Shephard, N., 1999. Filtering via simulation: auxiliary particle filters. J. Am. Stat. Assoc. 94 (446), 590–599.
Pratas, F., Trancoso, P., Stamatakis, A., Sousa, L., 2009. Fine-grain parallelism using multi-core, Cell/BE, and GPU systems: accelerating the phylogenetic likelihood function. In: 2009 International Conference on Parallel Processing. IEEE, pp. 9–17.
Pütz, B., Kam-Thong, T., Karbalai, N., Altmann, A., Müller-Myhsok, B., 2013. Cost-effective GPU-grid for genome-wide epistasis calculations. Methods Inf. Med. 52 (01), 91–95.
R Core Team, 2021. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. URL: https://www.R-project.org/.
Rajapaksa, S., Rasanjana, W., Perera, I., Meedeniya, D., 2019. GPU accelerated maximum likelihood analysis for phylogenetic inference. In: Proceedings of the 2019 8th International Conference on Software and Computer Applications, pp. 6–10.
Räss, L., Kolyukhin, D., Minakov, A., 2019. Efficient parallel random field generator for large 3-D geophysical problems. Comput. Geosci. 131, 158–169.
Redmon, J., 2013–2016. Darknet: open source Neural Networks in C. http://pjreddie.com/darknet/.
Roberts, G.O., Rosenthal, J.S., 2009. Examples of adaptive MCMC. J. Comput. Graph. Stat. 18 (2), 349–367.
Royle, J.A., Young, K.V., 2008. A hierarchical model for spatial capture–recapture data. Ecology 89 (8), 2281–2289.
Rubinpur, Y., Toledo, S., 2022. Signal processing for a reverse-GPS wildlife tracking system: CPU and GPU implementation experiences. Concurrency and Computation: Practice and Experience 34 (14), e6506.
Russell, D., Duck, C., Morris, C., Thompson, D., 2016. Independent estimates of grey seal population size: 2008 and 2014. Special Committee on Seals. Briefing Paper 16 (03), 79–87.
Sanders, J., Kandrot, E., 2010. CUDA by example: an introduction to general-purpose GPU programming, portable documents. Addison-Wesley Professional.
Sandric, I., Ionita, C., Chitu, Z., Dardala, M., Irimia, R., Furtuna, F.T., 2019. Using CUDA to accelerate uncertainty propagation modelling for landslide susceptibility assessment. Environ. Model Softw. 115, 176–186.
Sherlock, C., Thiery, A., Golightly, A., 2015. 'Efficiency of delayed-acceptance random walk Metropolis algorithms', *arXiv preprint arXiv:1506.08155*.
Sipos, I.R., Ceffer, A., Horváth, G., Levendovszky, J., 2019. Parallel MCMC sampling of AR-HMMs for prediction based option trading. Algorithmic Fin. 8 (1–2), 47–55.
Special Committee on Seals, 2021. Scientific advice on matters related to the Management of Seal Populations: 2021. Retrieved from. http://www.smru.st-andrews.ac.uk/files/2022/08/SCOS-2021.pdf.
Sreepathi, S., Kumar, J., Mills, R.T., Hoffman, F.M., Sripathi, V., Hargrove, W.W., 2017. Parallel multivariate spatio-temporal clustering of large ecological datasets on hybrid supercomputers. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp. 267–277.
Stone, J.E., Gohara, D., Shi, G., 2010. OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. 12 (3), 66–73.
Strnad, D., Kohek, Š., Kolmanič, S., 2018. Fuzzy modelling of growth potential in forest development simulation. Eco. Inform. 48, 80–88.
Suchard, M.A., Wang, Q., Chan, C., Frelinger, J., Cron, A., West, M., 2010. Understanding GPU programming for statistical computation: studies in massively parallel massive mixtures. J. Comput. Graph. Stat. 19 (2), 419–438.
Šukys, J., Kattwinkel, M., 2017. SPUX: scalable particle markov chain Monte Carlo for uncertainty quantification in stochastic ecological models. In: Bassini, S., Danelutto, M., Dazzi, P., Joubert, G.R., Peters, F. (Eds.), Advance, Parallel Computing in Everywhere, pp. 159–168.
Terenin, A., Dong, S., Draper, D., 2019. GPU-accelerated Gibbs sampling: a case study of the horseshoe Probit model. Stat. Comput. 29, 301–310.
Thomas, D.B., Howes, L., Luk, W., 2009. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In: Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, pp. 63–72.
Thomas, L., Russell, D.J., Duck, C.D., Morris, C.D., Lonergan, M., Empacher, F., Thompson, D., Harwood, J., 2019. Modelling the population size and dynamics of the British grey seal. Aquat. Conserv. Mar. Freshwat. Ecosyst. 29, 6–23.
Tuomanen, B., 2018. Hands-on GPU Programming with Python and CUDA: Explore High-Performance Parallel Computing with CUDA. Packt Publishing Ltd.
Vacondio, R., Dal Palù, A., Mignosa, P., 2014. GPU-enhanced finite volume shallow water solver for fast flood simulations. Environ. Model Softw. 57, 60–75.
Van Hemert, J.L., Dickerson, J.A., 2011. Monte Carlo randomization tests for large-scale abundance datasets on the GPU. Comput. Methods Prog. Biomed. 101 (1), 80–86.
Wang, Y., Mazur, T.R., Green, O., Hu, Y., Li, H., Rodriguez, V., Wooten, H.O., Yang, D., Zhao, T., Mutic, S., Li, H.H., 2016. A GPU-accelerated Monte Carlo dose calculation platform and its application toward validating an MRI-guided radiation therapy beam model. Med. Phys. 43 (7), 4040–4052.
Weiss, R.M., 2013. Accelerating swarm intelligence algorithms with GPU-computing. In: GPU Solutions to Multi-scale Problems in Science and Engineering. Springer, pp. 503–515.
Welch, M., Kwan, P., Sajeev, A., 2013. A high performance, agent-based simulation of old world screwworm fly lifecycle and dispersal using a graphics processing unit (GPU) platform. In: MODSIM 2013: 20th International Congress on Modelling and Simulation-Adapting to Change: The Multiple Roles of Modelling', Modelling and Simulation Society of Australia and new Zealand (MSSANZ).
White, G., Porter, M.D., 2014. GPU accelerated MCMC for modeling terrorist activity. Comput. Stat. Data Analysis 71, 643–651.
Whiteley, N., Lee, A., Heine, K., 2016. On the Role of Interaction in Sequential Monte Carlo Algorithms.
Wu, Y., Price, B., Isenegger, D., Fischlin, A., Allgöwer, B., Nuesch, D., 2006. Real-time 4D visualization of migratory insect dynamics within an integrated spatiotemporal system. Eco. Inform. 1 (2), 179–187.
Xue, P., Li, T., Zhao, K., Dong, Q., Ma, W., 2016. Glda: parallel gibbs sampling for latent dirichlet allocation on gpu. In: Advanced Computer Architecture: 11th Conference, ACA 2016, Weihai, China, August 22-23, 2016, Proceedings 11. Springer, pp. 97–107.
Zhou, C., Lang, X., Wang, Y., Zhu, C., 2015. gPGA: GPU accelerated population genetics analyses. PLoS One 10 (8), e0135028.
Zhou, Y., Donald, B.R., Zeng, J., 2017. Parallel computational protein design. Comput. Protein Design 265–277.