Ecological Modelling 222 (2011) 3011–3019

Contents lists available at ScienceDirect

Ecological Modelling

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

Scaling-up spatially-explicit ecological models using graphics processors

Johan van de Koppel[a,*], Rohit Gupta[a,b], Cornelis Vuik[b]

[a] Spatial Ecology Department, The Netherlands Institute of Ecology (NIOO-KNAW), Post Office Box 140, 4400 AC Yerseke, The Netherlands
[b] Delft University of Technology, Faculty EEMCS, Delft Institute of Applied Mathematics, 07.060, Mekelweg 4, 2628 CD, Delft, The Netherlands

ARTICLE INFO

Article history:
Received 7 September 2010
Received in revised form 24 April 2011
Accepted 6 June 2011
Available online 19 July 2011

Keywords:
CUDA
Graphics processors
Scaling laws
Self-organization
Spatially explicit models

ABSTRACT

How the properties of ecosystems relate to spatial scale is a prominent topic in current ecosystem research. Despite this, spatially explicit models typically include only a limited range of spatial scales, mostly because of computing limitations. Here, we describe the use of graphics processors to efficiently solve spatially explicit ecological models at large spatial scale using the CUDA language extension. We explain this technique by implementing three classical models of spatial self-organization in ecology: a spiral-wave forming predator–prey model, a model of pattern formation in arid vegetation, and a model of disturbance in mussel beds on rocky shores. Using these models, we show that the solutions of models on large spatial grids can be obtained on graphics processors with up to two orders of magnitude reduction in simulation time relative to normal pc processors. This allows for efficient simulation of very large spatial grids, which is crucial for, for instance, the study of the effect of spatial heterogeneity on the formation of self-organized spatial patterns, thereby facilitating the comparison between theoretical results and empirical data. Finally, we show that large-scale spatial simulations are preferable over repetitions at smaller spatial scales in identifying the presence of scaling relations in spatially self-organized ecosystems. Hence, the study of scaling laws in ecology may benefit significantly from implementation of ecological models on graphics processors.

1. Introduction

In the past years, spatially-explicit mathematical models have played an important role in the development of ecological theory (Levin, 1992; Rohani et al., 1997; Rietkerk and Van de Koppel, 2008). They have been used to gain understanding of the importance of space in the dynamics and conservation of species in both continuous (Hassell et al., 1994; Solé and Bascompte, 2006) and fragmented habitats (Hanski, 1994; Gravel et al., 2010). Especially models of spatial self-organization have been used to implement complexity theory principles to ecological systems. These models have been applied to for instance, spatial wave formation in large-scale predator-prey systems (Dunbar, 1983), to explain the formation of regular, self-organized spatial patterns in arid vegetation (Klausmeier, 1999), and scale-free patterns in disturbance-governed rocky intertidal mussel beds (Guichard et al., 2003). The basic premise behind self-organization models is that small-scale interactions between organisms generate coherent spatial patterns at larger spatial scales (Levin, 1992). Predicted patterns involve regular banded, dotted or labyrinth-shaped patterns that have been observed in many ecosystems all over the world (Rietkerk and Van de Koppel, 2008), and "power law" patterns that have universal characteristics over an extensive range of spatial scales, e.g., are scale-free (Pascual and Guichard, 2005). As theoretical models predict consistent changes in spatial patterns with changing environmental conditions, self-organized spatial patterns have been put forward as promising predictors of imminent degradation and desertification in ecosystems in response to global change (Rietkerk et al., 2004; Kefi et al., 2007b; Solé, 2007; Scheffer et al., 2009).

Despite the common use of spatial models to scale-up the effects of ecological interactions, most spatially-explicit models cover only a limited range of spatial scales. Partial differential equations that form the basis for most models of regular patterns are typically implemented on rectangular grids of 100×100 to 250×250 nodes. This allows a basic, limited scale prediction of regular pattern formation, but does not allow for the analysis of patterns along intrinsic or imposed gradients, modeling of nested patterns, or inclusion of evolutionary dynamics (because of the time scale differences between ecological and evolutionary processes). Cellular automaton models can be implemented over larger scales, but rarely exceed a size of 500×500 nodes. Yet, accurate fits of power law-shaped size-frequency distributions to describe patch formation depend crucially on precise predictions of the frequency of occurrence of large clusters or patches, which are often limited by the size of the simulated grid. Hence, to provide accurate predictions of the response of spatially self-organized systems to changing conditions, and to assess the value of the predicted

* Corresponding author. Tel.: +31 113 577455.
E-mail address: J.vandeKoppel@nioo.knaw.nl (J. van de Koppel).

0304-3800/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.ecolmodel.2011.06.004

patterns for indicator systems, predictions over extensive spatial scales are essential.

Here, we report on the use and implementation of graphics processors to efficiently solve spatially explicit ecological models on large spatial grids. Graphics processors, a common component of off-the-shelf multimedia computers, typically house a large number of processing cores that can efficiently process large grids, provided that the changes at each node (e.g., a point at the grid) can be predicted in parallel. This is the case for most spatial models where the entire grid is synchronously updated at the end of a single computing cycle (e.g., timestep). We describe how numerical algorithms are implemented on the graphics processor using the CUDA framework that is available for modern NVidia graphics cards. We provide example codes of implementations of three classical spatial models within ecological theory, the spiral-wave forming predator–prey model of Dunbar (1983), the model of self-organizing arid vegetation patterns by Rietkerk et al. (2002), and the mussel disturbance model by Guichard et al. (2003). We then study how the predictions of the models are affected by increased spatial scale.

2. Spatially explicit ecological models

Most ecological models described in theoretical ecology textbooks describe the changes in populations and the environment at a particular spatial location, or average their density over a particular spatial scale (e.g., a mean-field approach). Implicitly, these models assume that the heterogeneity that exists in natural systems at nearly every spatial scale is not of major influence to population dynamics. For many systems and for many modeling goals, this assumption does not hold. In these cases, the spatial structure of the population and the spatial movement of organisms and matter need to be modeled explicitly. Examples of this are the spatial dynamics of disease outbreaks, or the formation of regular spatial patterns in arid lands, which are described using spatially explicit models.

Spatially explicit models are often solved numerically using a spatial grid approach. At each node on the grid, both local population change and spatial fluxes are calculated as a function of population size at the node itself and the neighboring nodes (the scale of the neighborhood can be very local or more extensive). Often, the same calculations are repeated over the entire grid, calculating how population size at each node would change as described by a partial differential equation. In most modern computers, the rates of change are calculated for one node after the other until calculation has finished for the entire grid and population level at all nodes can subsequently be updated.

A special type of spatial model is the cellular automaton, which describes the spatial structure of any population using discrete states at each node, for instance being an individual of a specific species of plant. A node can change in state, for instance an individual plant being replaced by another, in a stochastic process, in which the replacement chance is a function of the state of the neighboring nodes. Similar to the partial-differential equation-based models, the same numerical operation is done for each node on the grid, calculating the new states based on the occupancies of the nodes in its current state.

3. Solving spatially explicit ecological models using a graphics processor

Normally, simulation models are executed by the computer's central processing unit, or CPU. Nearly all computers that are sold these days, however, also contain a graphics processing unit, or GPU, that handles the rendering of graphics before they are displayed on the screen. GPUs are either integrated within the main board of the computer, or placed on dedicated graphics cards. In particular GPUs on graphics cards are often very powerful computing chips. Rather than being able to do a broad range of tasks, which is the domain of the CPU, GPUs are specialized in floating point operations, e.g., calculations with numbers with many decimals. Another characteristic of modern GPUs is that they contain multiple cores, allowing the GPU to do many calculations at the same time. While current CPUs in personal computers contain between a single to four cores, high-end GPUs nowadays can contain many hundreds of cores per GPU, and hence can do a very high number of floating point calculations at the same time. These characteristics make graphics cards particularly suited to numerically solve spatially explicit models, especially those that are based on the above-described grid approach. The rates of change in several nodes can be calculated in parallel, which will significantly speed up the simulation.

The use of GPUs for parallel computation, for instance in solving spatially explicit ecological models, has to a large extent been limited by the difficulties of addressing them from mainstream computing languages such as C, Fortran or MatLab. Recently, GPU manufacturers are facilitating the use of graphics processors for custom programming applications by providing language extensions to standard programming languages. The mostly widely used extension at this moment is CUDA, the parallel computing architecture developed by NVidia for their GPUs. CUDA is an extension of C which can be used to execute the kernel of a simulation model (e.g., the part that gets repeated each time step) on the graphics processor. When a spatially explicit simulation model is implemented on the graphics processor, execution speed can increase between 10- and 100-fold as compared to a standard C implementation on the main processor, depending on the type of graphics card, the size of the grid and the type of spatial model.

Below, we first explain conceptually how a spatially explicit model is implemented on a graphics processor using CUDA. Then we provide three examples implementations of well-known spatially explicitly models from ecological theory to illustrate this technique, examples that can form an easy starting point for further model development. These example models are the predator–prey model by Dunbar (1983), the self-organizing arid vegetation model by Rietkerk et al. (2002), and the mussel bed disturbance model by Guichard et al. (2003). We briefly review these models, and then describe their implementation in CUDA. Note that the implementation of spatially explicit models on graphics processors, or in parallel processors in general, implies that all nodes are updated synchronously (e.g., at the same time). Asynchronous updating, where the nodes are updated in sequence (e.g., after each other), requires just a single thread, and is therefore best implemented on a single core, typically the CPU.

3.1. Implementing a spatially explicit model in CUDA

To run a spatially explicit simulation model on the GPU, it is crucial to understand how parallel processing of a spatial model is organized (Fig. 1). The calculations required to predict the rate of change at any node are called a thread. Each thread is processed by a specific processor core which has its own local memory to be used in calculations within the thread. However, the grids used in spatial models can easily have over 10,000 gridpoints, and hence even with the most extensive GPUs, each core has to handle multiple threads before the entire grid is updated. Haphazardly assigning processing cores to threads would be very inefficient. For this reason, the grid is split up in blocks, which is handled by what is called a multiprocessor, a group of 8 processor cores. This block has its own "shared" memory in which the variables needed to process the threads within this block are copied. Hence, the grid values

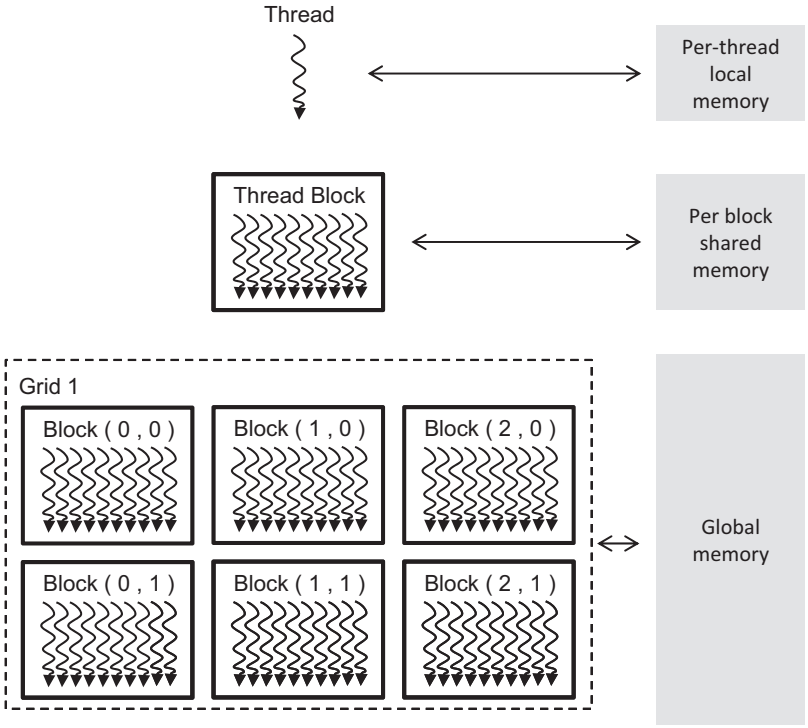

Fig. 1. Organization of computation and memory hierarchy on the GPU by the CUDA programming model. The computation of changes in the state of each node is called a thread. Threads are organized in blocks on which a multiprocessor operates, and all the thread blocks which make up the entire spatial grid. Each level has its own memory. See text for explanation.

used within this block are loaded into the multiprocessor's shared memory, allowing it to efficiently process all threads without additional access to the global GPU memory outside the multiprocessor, on which the entire grid is stored. The block structure ensures that the transfer of data from the graphics card's global memory to the graphics processor register memory is handled in an efficient way, preventing separate memory transfers for each individual thread. For best performance it is important that the information needed by each thread is local (e.g., from neighboring nodes), e.g., does not originate from the other side of the grid, so that it can all be loaded in the multiprocessor's memory before the block is executed.

When developing a model to run on the GPU, it is essential to optimize the size and number of the blocks, so that all processing cores are used efficiently. Obviously, a block size is limited by the size of the multiprocessor's memory, and hence large block sizes are not optimal. Beyond that, it is essential that the number of blocks is adjusted to the number of multiprocessors. For instance, if a GPU has 6 multiprocessors (e.g., 48 cores), it is inefficient to split the grid up into 8 blocks. Six blocks would then be first processed, after which two blocks remain. These would subsequently be processed by two multiprocessors, leaving the others waiting until those two have finished. It would be more efficient to divide the grid into 12 smaller blocks, so that after two rounds, all blocks have been processed and no multiprocessors remained idle for part of the time.

The number of multiprocessors and the size of their internal memory are specific to the type of GPU and graphics card; more expensive graphics cards have more multiprocessors and more memory. This means that in order to maximize the acceleration, the model needs to be geared to the GPU in the computer and recompiled before the model is run. However, a thread block size of 16×16 (256 threads) is a common choice that gave us good simulation results. More details about how to design an optimal configuration can be found in the NVidia CUDA programming guide at www.nvidia.com/cuda.

3.2. CUDA: an extension of the C language

Within any model program, the code needs to indicate which part of the program needs to be executed on the graphics processor. For this, a C language extension is provided that indicates that certain parts of the code need to be executed on the GPU, and that variables (most notably the arrays containing the state values at each node) and constants (e.g., the model's parameter values) need to be placed on the graphics processor before they are executed. For this, these variables have to be labeled specifically in the code using specific variable and function definitions that have been developed.

This is best explained using an example code. For this, we use Fisher's equation for a biological invasion of a species U in a spatial domain (Fisher, 1937):

$$\frac{\partial U}{\partial t} = U(1-U) + \frac{\partial^2 U}{\partial x^2}.$$

In this equation, local growth is modeled by the first term, the logistic growth equation with both intrinsic growth and maximum local density set to 1. Dispersal of organisms is modeled by the second term using a diffusion approximation. This model describes the formation of an invasion front that moves from one side of the domain to the other, starting with an initial condition where population density is one at the edge of the domain, and zero in the remaining space. We will describe the implementation of this model in CUDA below, assuming a basic understanding of the C programming language.

In Table 1, an example code is given that implements Fisher's equation along a one-dimensional domain U. This code describes how to implement the kernel that calculates the rate of change of local population density U in CUDA. For this, we need to copy the initial values of the domain U to the graphics processor, run the kernel that calculates the changes in U using the above equation for a number of times, and then copy the results back from the GPU memory to the CPU memory. We start with describing how the comput-

Table 1
A condensed example of a CUDA program, implementing Fisher's equation describing a biological invasion.

```
// includes, system & CUDA
#include <stdio.h>
#include <cuda.h>
// Parameter definitions
#define Block_Size    4                       // Thread block size
#define Block_Number 16                       // Number of blocks in the grid
#define Grid_Width (Block_Number*Block_Size) // Domain width
#define Length      100                       // The Length of the landscape
#define dT          0.1                       // Time step
/* ------- Device function that calculates U flux-------*/
__device__ float d2Udx2(float* U, int current) {
float dx, retval;
dx=(float)Length/Grid_Width;
retval = ( (-( U[current] - U[current-1] ) / dx )
          -(-( U[current+1] - U[current] ) / dx ) ) / dx ;
return retval;}
/* ------------ Main Simulation Kernel ----------------------*/
__global__ void dUdt_Kernel (float* U) {
int current;
current = blockIdx.x*Block_Size+threadIdx.x;
// The actual calculations for the model
if (current > 0 && current < Grid_Width-1) { // excluding the boundaries
     U[current]=U[current]+(U[current]*(1-U[current])+d2Udx2(U,current))*dT; }
__syncthreads(); } // End assembleMusselKernel
/* ---------- Program main ------------------------------------*/
int main(int argc, char** argv){
unsigned int size_U = Grid_Width;
unsigned int mem_size_U = sizeof(float) * size_U;
float* h_U, * d_U;
int time,x,i;
FILE *fp;
h_U = (float*) malloc(mem_size_U);   // allocate host memory
for(x=0;x<Grid_Width;x++) h_U[x]=0;  // initial values
h_U[0]=1;                            // the first domain point is one
/*--------- INITIALIZATION ON THE GPU ---------------------*/
// allocate device memory
cudaMalloc((void**) &d_U, mem_size_U);
// copy host memory to device
cudaMemcpy(d_U, h_U, mem_size_U,cudaMemcpyHostToDevice);
// setup execution parameters
dim3 threads(Block_Size);
dim3 grid(Grid_Width/ threads.x);
/*----- The simulation loop --------------------------------*/
for(time=1;time<500;time++) {
     dUdt_Kernel<<< grid, threads >>>(d_U); }
/*----- Printing the data and storing it in a file ----------*/
// copying back from device to host memory
cudaMemcpy(h_U, d_U, mem_size_U,cudaMemcpyDeviceToHost);
for(i=0;i<Grid_Width;i++)
    printf("%f ",h_U[i]);
printf("\r\n");
fp=fopen("Simple.txt","wt");
fwrite(&h_U,sizeof(float),Grid_Width,fp);
fclose(fp);
/*----- Cleaning up at the end of the program --------------*/
free(h_U);
cudaFree(d_U);
cudaThreadExit();
}
```

ing kernel that executes Fisher's equation is implemented in CUDA. First, any functions that need to be implemented on the kernel need to be specifically labeled. A function that is implemented on the GPU need to be declared as __global__; this we use de declare the function named dUdt_kernel. __device__ declares a function that is executed on the GPU and is called upon from another GPU-function (e.g., from the kernel that has been labeled by the __global__ instruction). This we use to implement the function d2Udx2, which calculates the dispersal term in Fisher's equation. The memory blocks that hold the gridded variables need to be defined as pointers, and then allocated dynamically (e.g., as the program runs) in the memory of the graphics card. After that, they can be filled from mirroring memory blocks on the main computer's memory. For these operations, CUDA has specific command called `cudaMalloc` and `cudaMemcopy`. The size of the grid that the graphics card can handle is obviously determined by the onboard memory of the graphics card. A card with 1 GB of memory will hold, for instance, 2 variables of type float with grid dimensions of 8192×8192, while a 4 GB Tesla card will hold 2 variables of $16{,}384 \times 16{,}384$ nodes. Note that an optimal speed is obtained using floating point variables of 4 bytes (e.g., type

float). When using double precision variables of 8 bytes (e.g., type double), most graphics cards are significantly slower.

Before the kernel is called upon, we need to specify the grid that the kernel will be working upon. In a normal C++ program, we would use a number of for statements to let the kernel move over the grid. When the model is being run on the GPU, we cannot define such a rigid structure, because a large number of cores will be processing the grid simultaneously. The programming that is needed for that is being taken care of by the CUDA compiler, and we just need to tell the compiler what the size of the grid is, and how many thread will be grouped within one block. This is done by defining two variables, called threads and grid, which follow a CUDA-defined type called dim3. Within the code, this would look like this for a 1D array:

```
dim3 threads (Block_Size);
dim3 grid (Width/threads.x);
```

Here, the names with the first letter capitalized are constants defined by the programmer. For 2 dimensional arrays, we provide three examples in the appendixes.

A CUDA implementation of Fisher's equation would, after including the standard input/output and CUDA header files (Table 1, lines 2–3) and defining the model's constants (lines 5–9), first define any specific functions that are used within the GPU kernel. In the case of Fisher's equation, we have defined a function called d2Udx2 that calculates the Laplacian operator $\partial^2 U/\partial x^2$, which is used in the kernel to calculate the rate of the dispersal of U (lines 11–16). This function is preceded by the `__device__` qualifier, indicating that it has to be run on and is only available for the GPU. After that, we define the main GPU kernel that calculates the rate of change of U for a specific location (lines 18–24). This function, called `dUdt_Kernel`, is preceded by the `__global__` qualifier, indicating that it is run on the GPU, but can be called upon the CPU part of the code (line 45). The position on the grid needs to be extracted using the `blockIdx.x` and `threadIdx.x` variables, which are provided by CUDA (line 20). The `__syncthreads()` command at the end of the kernel ensures that all the threads wait for every thread to have finished before the model continues, preventing non-parallel execution (line 24).

After this, the main program (lines 26–58) is defined, which specifies local variables (lines 27–31), allocates and initializes the gridded variables, and copies them to GPU memory (lines 32–39). The CUDA execution parameters are then defined and a for-loop is constructed that calls the GPU kernel for each timestep (lines 41–45; note: a loop in time, not in space). After that, the results are copied back from GPU to CPU memory, are displayed on the console and written to a file (lines 48–54). To finalize, all gridded memory blocks are freed from memory (lines 56–58).

It is beyond the scope of the paper to give a thorough description of the CUDA language definitions, for this we refer to the CUDA manual (NVidia CUDA programming guide, www.nvidia.com).

4. Application to spatially explicit ecological models

Here, we provide three example implementations of well-known ecological models for execution on the graphics processor. The models that we implement are the spiral-wave forming predator–prey model that was originally developed by Dunbar (1983), the model of self-organized pattern formation in arid vegetation by Rietkerk et al. (2002), and the stochastic mussel disturbance model by Guichard et al. (2003). The first two models are based on deterministic partial differential equations, where spatial exchange of organisms and matter drive the spatial dynamics of the system. The first model implements a predator–prey interaction, while the second model involves ecosystem-level processes in the form of water infiltration and overland flow. The last model is based on a stochastic cellular automaton, in which local changes in the presence or absence of mussels at any node is a function of the states of the neighboring nodes on a grid. We will briefly introduce these models, explain how they were implemented within CUDA, and then show how their performance and predictions are improved by solving them on the GPU.

4.1. Spiral waves in spatially explicit predator–prey systems

The first model that we will implement is the spatial predator–prey model that was initially developed by Dunbar (1983), and further studied by Hassell et al. (1991) in a host–parasite context. This model revealed the emergence of non-stationary spatial structures in the form of spatial waves and spirals, which allowed the persistence of otherwise non-persistent predator–prey interaction. The model describes the dynamics between a predator P and its prey N, as a function of their local interaction and the spatial movement of both species across the landscape. The equations of this model are given by:

$$\frac{\partial N}{\partial t} = N(1-N) - \frac{CN}{(1-CN)}P + \Delta N, \tag{1a}$$

$$\frac{\partial P}{\partial t} = \frac{P}{AB}\left(\frac{ACN}{1-CN} - 1\right) + \delta\Delta P. \tag{1b}$$

Here, parameters A, B and C represent dimensionless parameters that determine the strength of the interaction between the prey and the predator, the Laplacian operators ΔN and ΔP represent the dispersal of prey and predator, respectively, and the parameter δ represents the ratio between the movement rates of the predator and the prey. For a full explanation of this model, we refer to Sherratt et al. (2002).

4.2. Spatial self-organization in arid systems

An important insight that has been obtained from spatially explicit ecological models is that natural ecosystems can develop intricate spatial patterns in a process called spatial self-organization. An important real-world example of this process is the formation of tiger bush in arid ecosystems, where patches of shrub vegetation alternate with bare soil to form a regularly patterned landscape. The formation of this pattern can be explained by a spatial interaction between vegetation and water, the limiting resource in arid systems. Vegetation can promote the infiltration of rain water, promoting local growth conditions. Infiltration rates in patches with low to no vegetation are much lower, which can lead to accumulation of water on top of the soil, which induces exchanges of water between bare and vegetated patches in a process called overland flow. A well-known model that provides a mathematical description of this process is the model presented by HilleRisLambers et al. (2001) and Rietkerk et al. (2002):

$$\frac{\partial P}{\partial t} = cg_{\max}\frac{W}{W+k_1}P - dP + D_P\Delta P, \tag{2a}$$

$$\frac{\partial W}{\partial t} = \propto O\frac{P+k_2W_0}{P+k_2} - g_{\max}\frac{W}{W+k_1}P - r_WW + D_W\Delta W, \tag{2b}$$

$$\frac{\partial O}{\partial t} = R - \propto O\frac{P+k_2W_0}{P+k_2} + D_O\Delta O. \tag{2c}$$

Here, net plant growth $\partial P/\partial t$ is a function of water uptake $g_{\max}\cdot W/(W+k_1)\cdot P$ and losses $d\cdot P$ due to senescence or herbivory. Net change in soil water content $\partial W/\partial t$ is a function of infiltration $\alpha\cdot O\cdot(P+k_2\cdot W_0)/(P+k_2)$, water uptake by plants, and seepage $r_w\cdot W$. The net change in surface water $\partial O/\partial t$ is determined by rainfall R

and infiltration. The lateral fluxes of plants, soil water and surface water are modeled by a diffusion approximation, and are proportional to the Laplacian operators. A description of the parameters of this model is given in Rietkerk et al. (2002).

In Appendix B we provide a standard implementation of this model in CUDA on a flat surface. We have also implemented a heterogeneous landscape with this model of hills and valleys by varying the advection rate of water from 0 at flat surfaces to a maximal value on the hillsides. Moreover, we adopt higher water seepage into the deep soil ($r_w \cdot W$) on the hills than in the valleys, assuming a higher depth of the aquifer on the hills.

4.3. Disturbance driven self-organization in mussel beds

The last implementation of a spatial model that we provide is not based on partial differential equations, but on a cellular automaton. We implement the mussel disturbance model developed by Guichard et al. (2003), which describes the large-scale dynamics of mussel beds as a function of wave disturbance and recolonisation. At a small scale, each location can be in one of three states, being either filled with mussels, bare, or disturbed. The transitions between these states are described as a stochastic process, in which the chances of wave disturbance and recolonisation are a function of the presence of disturbed edges or other mussels in the direct neighborhood, respectively. For a detailed treatment see Guichard et al. (2003).

To compare CPU and GPU implementations of the above models, we used a HP z800 workstation with 2 Intel Xeon quad-core E5540 processors equipped with an NVidia Tesla C1060 computing processor with 240 cores and 4 GB of GDDR3 memory. The second system was a more modest Sony Vaio VPCF11Z1E Laptop with an Intel Core i7 720QM quadcore processor and an NVidia Geforce GT 330M with 48 cores and 1 GB of DDR3 memory. Comparisons were based on implementations of the above models, where the grid sizes were set to fit the memory of the Geforce GT 330M graphics card. All systems ran on Windows 7 with Microsoft Visual Studio 2008.

5. Results

Implementation of three classical ecological models revealed some clear advantages of the use of graphics processors for spatially explicit ecological models. The first advantage is the ability to run the models on extensive spatial grids. We present an example solution for each model in Fig. 2 at a grid size of 3072×3072 nodes. The cutouts in this figure demonstrate that a large spatial field can be simulated without losing detail at small spatial scales. The second advantage is that these simulations can be performed efficiently in a matter of seconds. We have made a comparison of the time it takes to solve the arid systems model on a grid of 1024×1024 nodes for 1000 timesteps, using MatLab as a baseline. Fig. 3 presents the simulation time, in s, of this model in MatLab and in C, both using only a single core, in Fortran using multiple cores, in CUDA on the low-end GPU, and in CUDA on the Tesla processor. A simulation which lasted 5400s in MatLab (1.5 h) on the HP z800 lasted only 15 s on the Tesla processor in the same machine, a 360 times speed increase. When more efficient use is made of the CPU using standard C or Fortran, a 200- (single-core C code) and 18 (multicore Fortran code)-fold speed increase is obtained still. This reveals the advantage that graphics processors provide over standard CPU processors. The advantage results from the many cores that GPU processors contain, as demonstrated by the Tesla computing card.

A possible use of the increased speed at which large spatial grids can be implemented on graphics cards is the possibility to simulate the effect of landscape heterogeneity on the process of

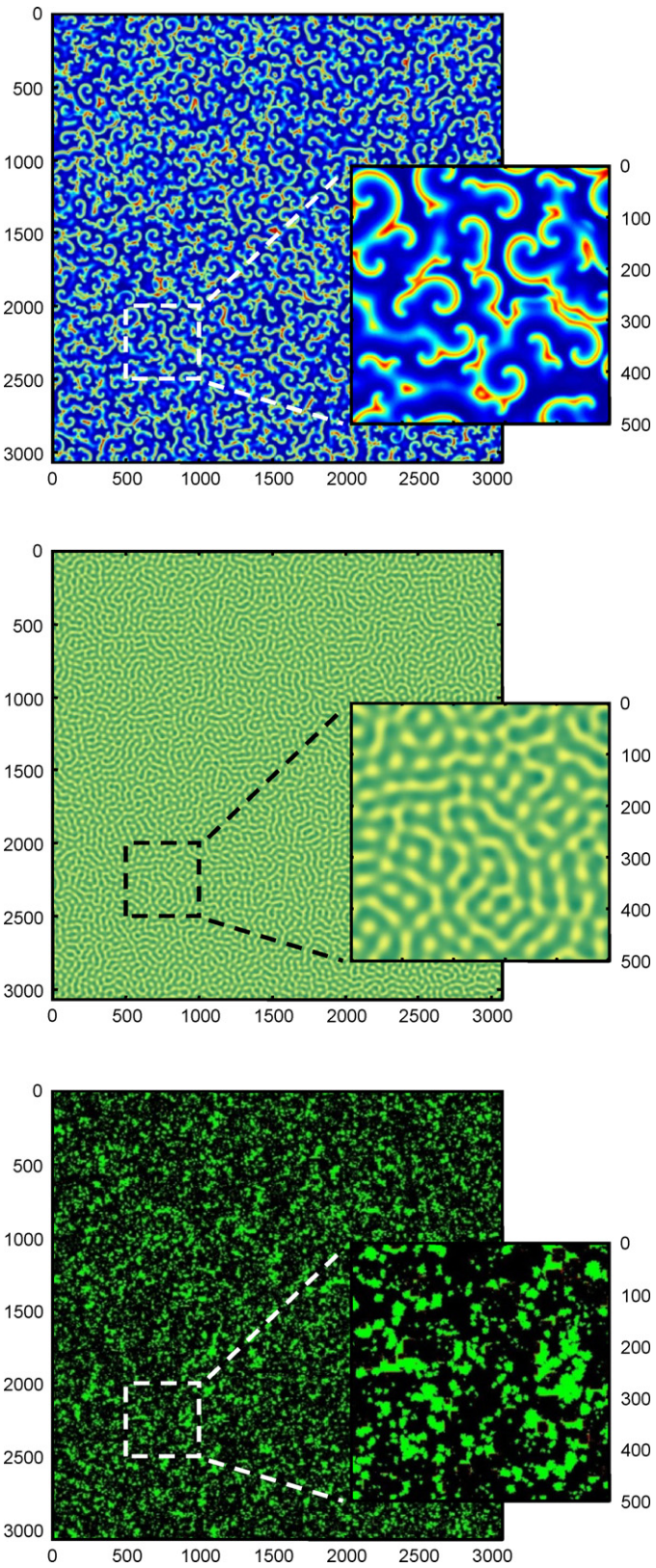

Fig. 2. Simulation output of (A) the predator–prey model, (B) the arid ecosystem model, and (C) the mussel disturbance model on grid of 3072×3072 nodes. The cutouts provide a model close-in view of 500×500 nodes.

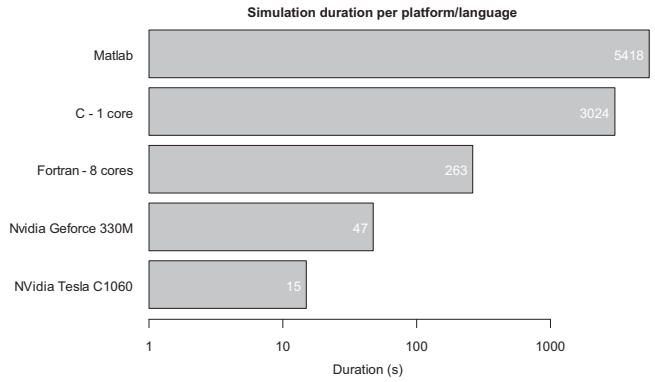

Fig. 3. Simulation times needed for the arid systems model on a grid of 1024 × 1024 nodes, implemented in MatLab (single thread), in C (single thread), in Fortran (8 threads), C-CUDA on NVidia Geforce 330M (48 threads), and in CUDA-C on NVidia Tesla C1060 (240 threads).

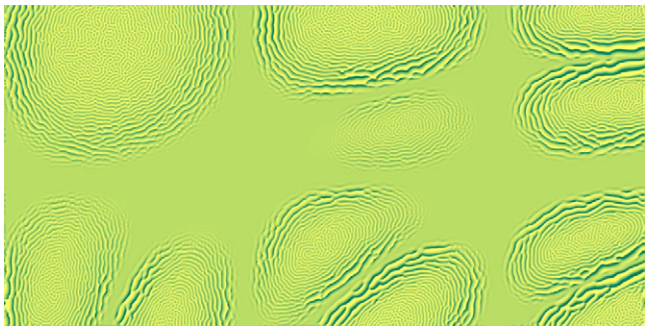

Fig. 4. Simulation of pattern formation in an arid ecosystem, positioned on a hilly landscape. The rounded shapes represent hills. In the valleys, no patterns develop as sufficient water is available, while on the hills, increased drainage reduces water availability which in turn causes pattern formation. On the hill slopes, this results in vegetation bands, as water flows directionally down to the lower parts. On the hilltops, labyrinth patterns develop, as water flows in all directions. This simulation represents a grid of 1024 × 2048 nodes.

spatial self-organization. We imposed a gently sloping landscape shape on the arid systems model. In this model, the advection rate of surface water is linearly related to the slope of the landscape. This affects patch formation in this landscape, which varies from round-shaped patches of limited size that are mostly caused by diffusion-approximated surface water movement on flat land, and larger, banded patches that occur on hillsides (Fig. 4). So far, the effects of changing slope have mostly been addressed in separate simulations. Using GPU processing, self-organization within heterogeneous landscapes can be studied efficiently, which greatly improves the possibility to compare the results of model simulations with real-world self-organized patterns.

5.1. Studying scaling relations

Cellular automaton models have formed the technical basis for the concept of criticality in ecology. This concept advances that spatially extensive ecosystems can exhibit threshold behavior when environmental conditions are changed. Near this threshold, the system is characterized by a scale-invariant mosaic of occupied and bare patches. Scale-invariance implies that the frequency of occurrence of patches of a particular size declines as a function of patch size itself, and appears as a linear relation when size and frequency are expressed on a logarithmic scale. Depending on the structure and assumptions of the models, scale-invariance of patch sizes is

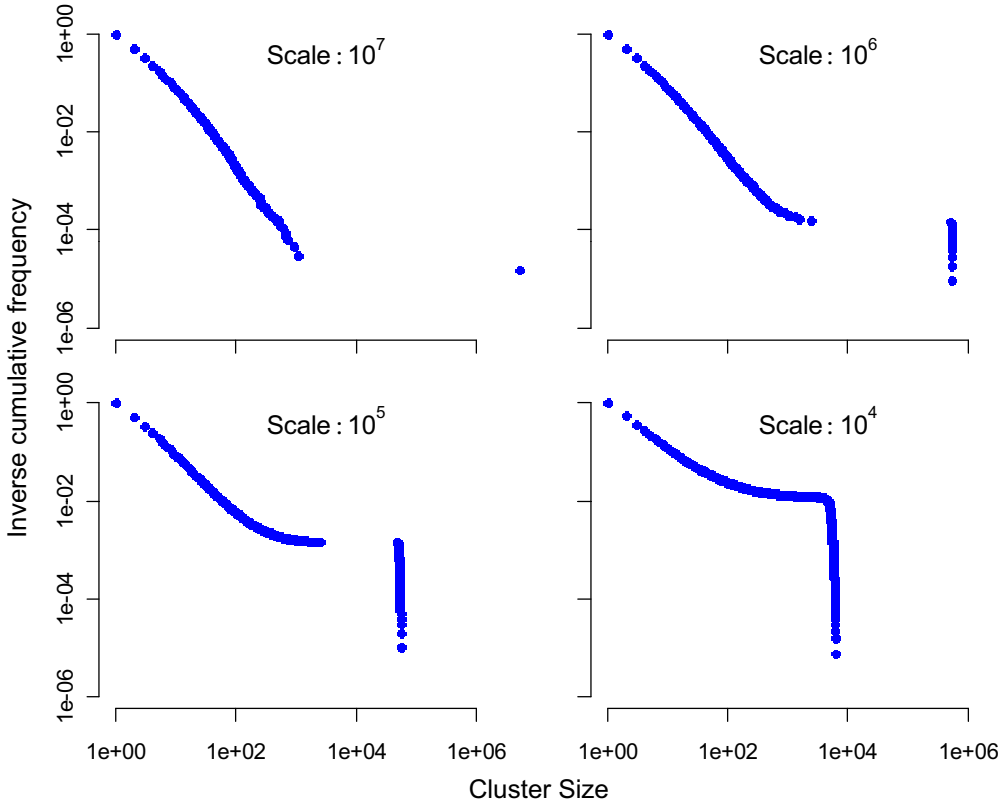

Fig. 5. The effects of scale on the predicted cluster size distribution in the mussel-disturbance model published by Guichard et al. (2003). For all 4 scales, a single predicted spatial pattern was used at scale 107, which was cut into pieces to obtain the lower scales. Note that in the largest spatial scale, a single large cluster is found that spans the entire grid. The remaining smaller patches follow a power law distribution. At smaller grid sizes, this spanning cluster is cut into pieces, forming a deviation at the high end of the cluster size distribution. Parameter values: $\alpha_0 = 0.4$, $\alpha_2 = 0.3$, the other parameters follow the above paper.

observed near the critical threshold (Malamud et al., 1998), is found for a broad range of conditions (Pascual and Guichard, 2005), or is actually absent near the threshold (Kefi et al., 2007a).

For successful identification of scale-invariance, it is crucial that a sufficiently large number of patches are included in the sample, in particular in the larger size classes. Most theoretical studies have therefore used a large number of simulations on relatively small grids of about 500×500 nodes to avoid problems with low sample sizes. Although this solves the problem of low sample sizes, the problem remains that the small size of the grid affects the presence of large patches, affecting the accuracy of the power law relation at the right of the curve. We illustrate this in Fig. 5. Using the GPU implementation of mussel disturbance model, we simulated the extensive field of 10^7 nodes (3162×3162), and analyzed the patch size frequency distribution following the procedure in Guichard et al. (2003). We then cut our grid into 9 parts of 10^6 nodes (1000×1000), and performed a patch size analysis on the combined grids. We repeated this procedure for two more times with grids of 10^5 and 10^4 nodes each. The results reveal that for this particular set of model parameter values, the results are significantly affected by the size of the grid. While grids of 10^4 to 10^6 nodes show a clear deviation from a straight curve, the grid of 10^7 reveals a clear power law relation. Hence, this result emphasizes the need for simulation of extensive spatial grids for accurate identification of scale-invariance in models of the spatial dynamics of ecosystems.

6. Discussion

In this paper, we demonstrate the use of graphics processors for simulation of spatial models of ecological self-organization. Graphics processing units, or GPUs, are processors that offload the rendering of 3D graphics from the computer's central processing unit (CPU). GPUs are specialized in floating point calculations. They contain many cores that can process calculations in parallel, and are therefore particularly suited for simulation of spatially explicit ecological models where the same calculation is repeated on a large spatial grid. This can lead to significant acceleration of the model, where spatial models are solved from 10- to over a 100-fold faster compared to calculations on the central processor unit (CPU). This acceleration makes it possible to simulate ecological models at large spatial scales while maintaining manageable simulation duration.

One of the leading themes in the field of spatial ecology is the ability of communities and ecosystems to self-organize. Two current developments in research on spatial-self-organization can benefit from GPU based techniques; the first being the use of self-organized spatial patterns as indicators of imminent catastrophic shifts (Rietkerk et al., 2004; Kefi et al., 2007a; Solé, 2007; Scheffer et al., 2009), and the second being the emergent properties of self-organized spatial patterns for ecosystem functioning (Van de Koppel and Rietkerk, 2004; Van de Koppel et al., 2005). Self-organized patterns are predicted to undergo a specific sequence of changes in response to changing environmental conditions before crossing a tipping point that leads to a dramatic and possibly irreversible shift. An important indicator is the disappearance of scale-invariance patchiness due to facilitation failure. As we demonstrate in this paper, identification of scale-invariance in theoretical studies can dependent critically on simulating the dynamics of spatial models at large spatial scales (Burroughs and Tebbens, 2002). When simulations are performed on small scales, deviations from power laws are found to be significant, while simulations at sufficiently large scale would point at scale-invariance.

We have introduced the use of CUDA in spatially explicit ecological models using examples of self-organized systems. Studies on spatial self-organization typically apply simulations within small spatial domains and homogeneous conditions (Rietkerk et al., 2002; Van de Koppel et al., 2005). Within many ecosystems, however, there are large-scale gradients in environmental conditions that are caused by the physical landscape, for instance on hillsides or mountainous terrain, or in intertidal ecosystems where fluxes of nutrients are not homogeneously spread over the area (as in case of rainfall) but enter from the boundaries via tidal flow, and are gradually depleted as the water moves off the tidal flat. These external or intrinsic gradients could significantly affect how ecosystems respond to sudden or gradual changes in environmental conditions, and hence affect the impact of emergent phenomenon on ecosystem functioning. To study the impact of these underlying or intrinsic gradients, models need to be simulated within extensive spatial domains with large grid sizes. Hence, using GPU-based techniques for solving spatially explicit models will provide a fast, cheap and relatively simple method (compared to the use of extensive computing clusters) for predicting the large-scale response using more realistic, heterogeneous ecosystem models.

In this paper, we have addressed how to implement reaction-diffusion models cellular automata on the GPU. However, it is well possible to implement other types of models, such as individual-based models (IBMs) or networks of meta-communities. The requirement for implementation of these models on the GPU is that on each individual or metacommunity, the same calculation is performed in parallel (e.g., not sequential). Individual-based models of schools of fish or migratory organisms have successfully been implemented on GPUs, as a similar movement algorithm is applied to all individuals (Erra et al., 2009; Guttal and Couzin, 2010). It is most efficient from computational viewpoint if individuals that interact with each other in the simulated group (e.g., are in each other's neighborhood) are also in proximity in the computer's memory, the individuals need to be continuously reordered in the computer's memory to match changes in the organisms coordinates (Erra et al., 2009). This required a complex reordering algorithm, and hence implementation of these IBM models on GPUs is far more complex that the grid models that we discussed here.

As it is a new development, programming an ecological model to run on the GPU requires some knowledge of low-level programming in C, as was described in this paper. As the use of GPUs is becoming more commonplace, high-level language support is rapidly expanding. Specialized computing algorithms, for instance for calculating a Fast-Fourier-Transformation of a extensive grid of data, can be executed on the graphics processor from programming environments like MatLab and Python using language extension packages. Moreover, high-level support for the use of GPUs within program languages like MatLab and PGI Fortran and in mathematical applications such as Mathematica has recently become available. Hence, although current use of GPUs in ecological modeling requires in-depth understanding the CUDA programming model, GPU programming is likely to become more accessible to ecological modelers in the near future.

Another recent development is the adoption of OpenCL as a common standard for writing programs that run across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL, also based on C, can be used to implement models on GPUs of both NVidia and AMD (or ATI), and can be used on both windows-based and Apple computers. Drawbacks of OpenCL are that it requires more extensive coding, is currently less mature (e.g., less tools are available) and requires the programmer to put more effort in memory management. We therefore consider CUDA to be a better choice for ecological modelers that take their first steps in GPU programming, but this may change in the future.

The CUDA implementations that we present in Appendix B have been optimized for clarity, not for efficiency. Further improvements

in efficiency can be obtained using shared memory algorithms, where thread blocks that are copied from global memory into faster shared memory are reused efficiently before global memory is accessed again (see the NVidia CUDA programming guide on www.nvidia.com/cuda). This can lead to a further 10–20% speed increase in the models we presented here. GPU programming can further be combined with more traditional solving techniques such as Runge–Kutta integration to further improve efficiency, as long as these techniques use explicit iterations to approximate the model's solution. Discussing these methods in detail is beyond the scope of the current paper.

It is now common knowledge that most properties of ecological and biological systems depend strongly on the scale at which they are studied. This has led to a broad theory that considers how spatial scale affects observations and how changes in the characteristics of ecological systems with spatial scale can reflect ecological processes (Peterson and Parker, 1998). When modeling the dynamics of spatially explicit ecosystems, however, the effects of scale are often ignored, and the range of scales at which ecological processes are modeled is typically limited. This has mostly been caused by technical limitations, as simulating large spatial grids requires extensive computing facilities. GPU based techniques allow mathematical ecologists simulate the dynamics of ecological systems on large spatial grids, allowing for a range of spatial scales to be addressed. These techniques will significantly facilitate mechanistic understanding of scale-dependence and scale-invariance in ecological systems.

Acknowledgement

We would like to thank Fred Guichard for comments on a draft of the manuscript.

Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.ecolmodel.2011.06.004.

References

Burroughs, S.M., Tebbens, S.F., 2002. The upper-truncated power law applied to earthquake cumulative frequency-magnitude distributions: evidence for a time-independent scaling parameter. Bull. Seismol. Soc. Am. 92, 2983–2993.
Dunbar, S.R., 1983. Traveling wave solutions of diffusive Lotka–Volterra equations. J. Math. Biol. 17, 11–32.
Erra, U., Frola, B., Scarano, V., Couzin, I., Soc, I.C., 2009. An efficient GPU implementation for large scale individual-based simulation of collective behavior. IEEE Comput. Soc. Los Alamitos, 51–58.
Fisher, R.A., 1937. The wave of advance of advantageous genes. Ann. Eugen. 7, 353–369.
Gravel, D., Mouquet, N., Loreau, M., Guichard, F., 2010. Patch dynamics, persistence, and species coexistence in metaecosystems. Am. Nat. 176, 289–302.
Guichard, F., Halpin, P.M., Allison, G.W., Lubchenco, J., Menge, B.A., 2003. Mussel disturbance dynamics: signatures of oceanographic forcing from local interactions. Am. Nat. 161, 889–904.
Guttal, V., Couzin, I.D., 2010. Social interactions, information use, and the evolution of collective migration. Proc. Natl. Acad. Sci. U.S.A. 107, 16172–16177.
Hanski, I., 1994. Spatial scale, patchiness and population-dynamics on land. Philos. Trans. R. Soc. Lond. B: Biol. Sci. 343, 19–25.
Hassell, M.P., Comins, H.N., May, R.M., 1991. Spatial structure and chaos in insect population-dynamics. Nature 353, 255–258.
Hassell, M.P., Comins, H.N., May, R.M., 1994. Species coexistence and self-organizing spatial dynamics. Nature 370, 290–292.
HilleRisLambers, R., Rietkerk, M., van den Bosch, F., Prins, H.H.T., de Kroon, H., 2001. Vegetation pattern formation in semi-arid grazing systems. Ecology 82, 50–61.
Kefi, S., Rietkerk, M., Alados, C.L., Pueyo, Y., Papanastasis, V.P., ElAich, A., de Ruiter, P.C., 2007a. Spatial vegetation patterns and imminent desertification in mediterranean arid ecosystems. Nature 449, 213–215.
Kefi, S., Rietkerk, M., van Baalen, M., Loreau, M., 2007b. Local facilitation, bistability and transitions in arid ecosystems. Theor. Popul. Biol. 71, 367–379.
Klausmeier, C.A., 1999. Regular and irregular patterns in semiarid vegetation. Science 284, 1826–1828.
Levin, S.A., 1992. The problem of pattern and scale in ecology. Ecology 73, 1943–1967.
Malamud, B.D., Morein, G., Turcotte, D.L., 1998. Forest fires: an example of self-organized critical behavior. Science 281, 1840–1842.
Pascual, M., Guichard, F., 2005. Criticality and disturbance in spatial ecological systems. Trends Ecol. Evol. 20, 88–95.
Peterson, D.L., Parker, V.T. (Eds.), 1998. Ecological Scale: Theory and Applications. Columbia University Press, New York.
Rietkerk, M., Boerlijst, M.C., Van Langevelde, F., HilleRisLambers, R., Van de Koppel, J., Kumar, L., Klausmeier, C.A., Prins, H.H.T., De Roos, A.M., 2002. Self-organisation of vegetation in arid ecosystems. Am. Nat. 160, 524–530.
Rietkerk, M., Dekker, S.C., de Ruiter, P.C., van de Koppel, J., 2004. Self-organized patchiness and catastrophic shifts in ecosystems. Science 305, 1926–1929.
Rietkerk, M., Van de Koppel, J., 2008. Regular pattern formation in real ecosystems. Trends Ecol. Evol. 23, 169–175.
Rohani, P., Lewis, T.J., Grunbaum, D., Ruxton, G.D., 1997. Spatial self-organization in ecology: pretty patterns or robust reality? Trends Ecol. Evol. 12, 70–74.
Scheffer, M., Bascompte, J., Brock, W.A., Brovkin, V., Carpenter, S.R., Dakos, V., Held, H., van Nes, E.H., Rietkerk, M., Sugihara, G., 2009. Early-warning signals for critical transitions. Nature 461, 53–59.
Sherratt, J.A., Lambin, X., Thomas, C.J., Sherratt, T.N., 2002. Generation of periodic waves by landscape features in cyclic predator–prey systems. Proc. R. Soc. Lond B: Biol. Sci. 269, 327–334.
Solé, R., 2007. Ecology: scaling laws in the drier. Nature 449, 151–153.
Solé, R.V., Bascompte, J., 2006. Self-organization in Complex Ecosystems. Princeton University Press, Princeton, NJ.
Van de Koppel, J., Rietkerk, M., 2004. Spatial interactions and resilience in arid ecosystems. Am. Nat. 163, 113–121.
Van de Koppel, J., Rietkerk, M., Dankers, N., Herman, P.M.J., 2005. Scale-dependent feedback and regular spatial patterns in young mussel beds. Am. Nat. 165, E66–E77.