Open access • Journal Article • DOI:10.1007/S10287-015-0246-X

Accelerating viability kernel computation with CUDA architecture: application to bycatch fishery management — Source link

Antoine Brias, Jean-Denis Mathias, Guillaume Deffuant

Institutions: Local Initiatives Support Corporation

Published on: 02 Jan 2016 - Computational Management Science (Springer Berlin Heidelberg)

Topics: General-purpose computing on graphics processing units, CUDA and Kernel (image processing)

Accelerating viability kernel computation with CUDA architecture: application to bycatch fishery management

A. Brias, Jean-Denis Mathias, G. Deffuant

Noname manuscript No.
(will be inserted by the editor)

Accelerating viability kernel computation with CUDA architecture : application to bycatch fishery management

Antoine Brias · Jean-Denis Mathias · Guillaume Deffuant

Received: date / Accepted: date

Abstract Computing a viability kernel consumes time and memory resources which increase exponentially with the dimension of the problem. This curse of dimensionality strongly limits the applicability of this approach, otherwise promising. We report here an attempt to tackle this problem with Graphics Processing Units (GPU). We design and implement a version of the viability kernel algorithm suitable for General Purpose GPU (GPGPU) computing using Nvidia's architecture, CUDA (Computing Unified Device Architecture). Different parts of the algorithm are parallelized on the GPU device and we test the algorithm on a dynamical system of theoretical population growth. We study computing time gains as a function of the number of dimensions and the accuracy of the grid covering the state space. The speed factor reaches up to 20 with the GPU version compared to the Central Processing Unit (CPU) version, making the approach more applicable to problems in 4 to 7 dimen-

A. Brias
Irstea, UR LISC Laboratoire d'ingénierie des systèmes complexes, 24 avenue des Landais-BP 20085 Aubière, F-63172 France
Tel.:+123-45-678910
Fax: +123-45-678910
E-mail: antoine.brias@irstea.fr

sions. We use the GPU version of the algorithm to compute viability kernel of bycatch fishery management problems up to 6 dimensions.

Keywords Viability kernel · Dynamic programming · CUDA · GPU · Fishery management

Mathematics Subject Classification (2000) 90B50 - Management Decision Making, including Multiple Objectives · 90C39 - Dynamic Programming

1 Introduction

Viability theory provides mathematical and numerical tools and concepts for maintaining a dynamical system within a set of states (called the constraint set). This theory has numerous potential applications in food processing, finance, economics, environment (Sicard et al, 2012; Béné et al, 2001; Doyen et al, 2012; Bernard and Martin, 2013; Andrés-Domenech et al, 2014; Chapel et al, 2008, 2010; Mathias et al, 2015). A essential step in this approach lies in the computation of the viability kernel. This is the set of states for which there exists a control policy that keeps the system in the constraint set for some (finite or infinite) time. Since the 90's, several algorithms have been developed to compute viability kernels in different application fields (Saint-Pierre, 1990; Bokanowski et al, 2006). Some of them are adapted to finite horizon problems (Djeridane and Lygeros, 2008), linear dynamics (Kaynama and Oishi, 2013) or particular kinds of problems such as single output nonlinear control systems affine in the control (Turriff and Broucke, 2009; Mattioli and Artiouchine, 2003). The concepts have been extended to the stochastic case (Doyen and De Lara, 2010; Rougé et al, 2013), in which dynamics includes random variables accounting for uncertainties. However, most of them are limited by their computational complexity. When the number of dimensions increases,

the required computation time and memory increase exponentially because they are mainly based on a grid covering the state space.

This curse of dimensionality is a moot point in a number of domains, like data mining, numerical analysis (Donoho, 2000) and multiple approaches aim at mitigating it and at speeding up viability kernel computing. For instance, Bonneuil (2006) attempts to handle large dimensional state space with computing control policies on a given time horizon with simulated annealing. Deffuant et al (2007) use Support Vector Machines for approximating viability kernels. Maidens et al (2009) try to overcome this curse by using Lagrangian methods, which do not call for a grid. It offers particularly interesting performances for linear dynamics. Designing new algorithms is a crucial way to reduce the curse of dimensionality, but existing technological solutions can also lead to important gains of time. Some programs like Vikaasa, used in by-cash fishery management, embed a multi-core implementation of viability kernel computation (Krawczyk et al, 2013). However, in recent years, another technical improvement arose which seems well adapted for the viability kernel problematic : the GPU parallel computing technology. Historically, GPUs are electronic components used in computer graphics hardware. They are getting more widely used to accelerate computations in many fields (Mametjanov et al, 2012; Cekmez et al, 2013; Sabo et al, 2014). Due to its architecture, the GPUs solution is advantageous for problems for which a parallel execution is possible (Goldsworthy, 2014). It often exceeds the capacity of computing clusters in terms of performance and for a lower cost. Thus, NVIDA has created the CUDA platform to offer tools and to allow programmers to use the parallel architecture of graphic cards. The basic point is to separate the program in two parts : the serial code executed by the host (the Central Processing Unit (CPU)), and the parallel code, executed in the GPU threads across multiple processing elements.

In this paper, we use the standard viability algorithm (Saint-Pierre, 1990) which can be considered as a particular case of dynamic programming. To illustrate our work, the implemented algorithm is tested on a basic multidimensional model of population dynamics. In particular, we compare the computation time between the parallelized version and the sequential one. Finally, the algorithm is used to compute the viability kernel in a case of bycatch fishery management.

2 Viability theory and dynamic programming

2.1 Viability kernel

We consider a discrete dynamics in which h is the function mapping the state and the control at time t with the state at the next time step $t+1$:

$$x\left(t+1\right)=h\left(x\left(t\right),u\left(t\right)\right) \tag{1}$$

with $x\left(t\right)\in\mathcal{X}$ the set of states, $u\left(t\right)\in\mathcal{U}\left(x\left(t\right)\right)$ which is a finite set of possible controls allowing the regulation of the dynamics and $h\left(.,.\right):\mathcal{X}\times\mathcal{U}\rightarrow\mathcal{X}$, associating a state $x\left(t\right)\in\mathcal{X}$ and a control $u\left(t\right)\in\mathcal{U}\left(x\left(t\right)\right)$ with a successor $x\left(t+1\right)\in\mathcal{X}$. For example, it associates the closest grid point of the successor obtained with an Eulerian scheme defined with an appropriate local parameter. We want to assess the viability kernel $Viab_{T}\left(K\right)$; the set of initial states for which a control strategy exists to maintain the system inside K, the set of desirable states (the viability set), until T (a finite horizon). This is rewritten in Eq.2 :

$$Viab_{T}\left(K\right)=\left\{x_{0}\in K|\exists\left(u\left(0\right),u\left(1\right),...,u\left(T-1\right)\right),\forall t\in\left\{0,1,...,T\right\},x\left(t\right)\in K\right\} \tag{2}$$

with $x(t)$ the state of the system at date t and $u(t)$ the chosen control value at t, which maintains the system in the viability set K.

Different algorithms are available for computing this set. Because of the approximation of the dynamical system on a grid, these algorithms are submitted to the dimensionality curse: the necessary time and memory increase exponentially with the number of dimensions of the problem. We choose to focus on a dynamic programming algorithm which is very standard and well suited for a parallel approach.

2.2 Dynamic programming

We assume that the dynamics of the system fulfils the condition ensuring the convergence of the approximation to the actual kernel, when the resolution of the discretization tends to 0 (Saint-Pierre, 1990). The dynamic programming algorithm is a backward recursive algorithm and determines a value function : $V(t,x)$ at date t. The seed value at T is :

$$V(T,x) = \begin{cases} 1 & \text{if } x \in K \\ 0 & \text{if } x \notin K \end{cases} \tag{3}$$

In this case, the value function directly determines the viability kernel. At T, being viable is the property of belonging to the viability kernel. The linear recurrence relation is given by :

$$\forall t \in \{0, T-1\}, V(t,x) = \begin{cases} \max\limits_{u \in \mathcal{U}(x)} V(t+1, h(x,u)) & \text{if } x \in K \\ 0 & \text{if } x \notin K \end{cases} \tag{4}$$

Thus, at $t = 0$:

$$V(0,x) = \begin{cases} 1 & \text{if } x \in Viab_T(K) \\ 0 & \text{if } x \notin Viab_T(K) \end{cases} \tag{5}$$

The principles of dynamic programming are respected because, at each step time :

- each point of the grid is a subproblem (V has to be computed for each point);
- a value function has to be optimized independently for these points;

The value function V takes only two values : 0 and 1. We are looking for initial states, for which the value function at time horizon T is 1. Eq.4 is equivalent to say that the value function $V(t,x)$ is equal to 1 if a control $u \in \mathcal{U}(x)$ exists, maintaining the system in the constraint set K. Otherwise, the value function $V(t,x)$ equals to 0. The backward dynamic programming equation defines the value function $V(x,t)$. By writing a max instead of a sup, we implicitly assume the existence of an optimal solution for each time t and state x. The algorithm involves two main steps:

- For each point of the grid, we store all the successors (the point of the grid at the next time step) obtained for each choice of control (the possible set of actions at each point is supposed finite).
- The second part performs iterations of kernel approximations by progressively excluding points of the grid until reaching a fixed point or a predefined number of time steps.

Thus we have the algorithm Alg.1 as follows :

Algorithm 1 Detailed Dynamic Programming Viability Kernel Algorithm

$$\text{Storing successors} : \begin{cases} i = 0 \\ K^0 = K \\ \text{store coordinates, } \forall x \in K \\ \text{store } h\left(x,u\right), \forall x \in K, \forall \mathbf{u} \in \mathcal{U}\left(x\right) \end{cases}$$

$$\text{Iterating kernel approximation} : \begin{cases} \text{repeat} \\ \quad K^{i+1} = \left\{x \in K^i | V\left(i+1,x\right) = 1\right\} \\ \quad i = i+1 \\ \text{until } K^{i+1} = K^i \text{ OR } K^{i+1} = \emptyset \text{ OR } i = T \end{cases}$$

$$Viab_T\left(K\right) = K^i$$

2.3 Parallelizing dynamic programing algorithm

Dynamic programming is well suited to parallelization in the two main parts. During the storing of successors, all succcessors $x\left(t+1\right)$ of each state and each value of u, defined in Eq.1, can be computed separately. In the second part of the algorithm, it is possible to parallelize the update of the value function $V(t,x)$ applied to each state (Eq.4). In both cases, computations are completely independent for one step of time. The parallelizing of all these calculations thus improves the performance in terms of computation time. Since GPUs have hundreds of processor cores running in parallel, they can be more efficient than using traditional CPU architectures.

3 Using GPU for computing viability kernels

3.1 GPU architecture

As they are widely used in many scientific computing fields, graphics hardware programming for general purpose computing is well documented, this includes ways to write and debug the code (Langdon, 2011). Providing highly effective data parallelism, GPU cores contain multiple threads running at the same time

which can accelerate viability algorithms. The CUDA programming framework enables the use of all the cores belonging to the GPU for general purpose computing. Programs written in this paper use the C++ CUDA framework. A GPU computing view consists in splitting the input data between the different threads, and performing operations on these pieces of data, independently and thus achieving parallelization. This kind of computing encourages to rethink programs to make this parallelization possible. Fig.1 illustrates the architecture of a GPU device which is seen as a grid housing blocks. Each block involves a number of threads performing the computations. The GPU device owns a global memory, each block has a memory shared by all of its threads, and each thread owns a smaller local memory. Here, we use the global memory to store big arrays of data with which all the threads can interact. In a block, a thread is indexed over 3 dimensions (a block can be shown as a cube of threads), thus the indexation has to be carefully done. Each thread and each block are indexed at their level. In our implementation, we want to get the ID $threadID$ of each one-dimensional thread in a grid of 3D blocks. According to the CUDA framework writing convention , this ID is given by :

$$\begin{aligned} threadID =& threadIdx.i + blockDim.i * blockIdx.i + \\ & gridDim.i * blockIdx.j * blockDim.i + \\ & gridDim.i * gridDim.j * blockIdx.k * blockDim.i \end{aligned} \tag{6}$$

Here, $threadIdx.i$ means the coordinates i of $threadIdx$. $gridDim$ contains the dimensions of the grid (the grid owns $gridDim.i \times gridDim.j \times gridDim.k$ blocks), $blockIdx$ contains the block indexes within the grid (in 3 dimensions : $blockIdx.i$, $blockIdx.j$ and $blockIdx.k$), $blockDim$ contains the dimensions of the block (a block contains $blockDim.i \times blockDim.j \times blockDim.k$ threads

in 3 dimensions) and $threadIdx$ contains the thread indexes within the block (in 3 dimensions too, but for simplicity, only one is used here : $threadIdx.i$). These variables are mainly used in assigning the work per block and thread. The maximum values in each dimension in both $blockDim$ and $gridDim$ are GPU-dependent. At least, the i component needs to be declared. By default, the value for the j and k components is 1.

3.2 Overall strategy

The organization of the algorithm is summed up in Fig. 2. The figure shows the storing of trajectories and the iterating kernel approximation steps of Alg.1. The steps are :

- the initialization and setting of the GPU device (a);
- the initialization of the model parameters (b). The user inputs these parameters;
- the initialization of CUDA parameters (number of blocks, number of threads by blocks in each dimension)(c);
- the creation of empty arrays, which will contain controlled evolutions and the value function of each point using the CUDA instruction $cudaMalloc$ (d);
- the copy of parameters from host to the GPU device (e) using the CUDA instruction $cudaMemcpy$ with the flag $cudaMemcpyHostToDevice$;
- the computation of successors of each point i.e. the storing of next point of the trajectory for each possible value of the control Alg.1. This operation is parallelized on the GPU and it is done in the first kernel function. Section 3.3 detailed this part.
- the iterating kernel approximation step of Alg.1. The value function of each point is calculated and ran until the stop condition is reached. It is paral-

lelized in the second kernel function on the GPU. Section 3.4 detailed this part.

- the copy of results from the GPU memory to the CPU (f) using the CUDA instruction *cudaMemcpy* with the flag *cudaMemcpyDeviceToHost*;
- The clear of the memory of the GPU and the finalization of the GPU device (g), using the CUDA instruction *cudaFree*;
- the disallocation of the CPU memory (h).

3.3 Parallelizing the computing and the storing of successors

The input of the algorithm is the number of points by dimension and each dimension bound. The algorithm computes the coordinates of all the points and their index. It can be parallelized since each point is independent from the other. Thus, one thread on the GPU will compute the coordinates of one point.

Now, we have to compute the all the successors of each point of the grid, when applying the different values of the control (if the control can take continuous values then it is discretized). We can thus suppose that the control takes $nbControls$ values point of the grid. Then, on each thread, the coordinates of one state X_i are sent, and all the $nbControls$ possible evolutions from this state are returned. Again, the number of threads is equal to the number of points of the grid. This treatment is summed up on Fig. 3. There are four steps in the GPU computation of the trajectories :

- the inputs are the values of the discretized control, $U = (u_1, ..., u_{nbControls})$ and the coordinates of all the points $X = (X_1, ..., X_{nbPoints})$. These coordinates were determined beforehand and remain in the GPU global memory;
- this data is sent on threads, each thread holding the computation of all the controlled evolutions of only one point;

- each thread then sends back the result in the array H. This array contains the index of the controlled evolutions. Moreover, it contains -1 if the evolution doesn't belong to K. Thus, we store the index of i-th controlled evolution of X in $H(X,i)$. As the number of threads is the same for each block, some threads can be superfluous on the last block;
- once we get the controlled evolutions, we just have to copy them from the GPU to the CPU and we apply the second part of the dynamic programming algorithm.

3.4 Parallelizing the iterating kernel approximation

The second part of the algorithm computes the value function V for each point until the stopping condition is reached. As shown on Fig. 4, we parallelize this task. The global memory contains the value function array : v in which the value function V of each point is stored; the successors computed at the first step : H; and a boolean value : $CheckChange$ which states if there is no change during the step. In this latter case, the algorithm stops. The current value function V is stored in a temporary array $vTmp$. The treatment below (Alg. 2) is applied to each thread (one point X by thread) :

Algorithm 2 Update of the value function of X in the iterating kernel approximation.

If $v(X)$=1 (We check if the point is viable, i.e., if the value function applied to it is equal to 1)
    If $\forall i \in [1, nbControls - 1]$,
    $(H(X,i) = -1)$ OR $(H(X,i) \geq 0$ AND $v(H(X,i)) = 0)$
        $vTmp(X) = 0$
        $CheckChange = CheckChange + 1$
    End If
End If

If the point is viable and if a viable successor exists, nothing is done, the point is still viable. But if none of its successors is viable, i.e. if none of its successors belongs to K (i.e., $H(X,i) = -1$) or if the value function V of the successors is equal to 0, then the point is considered non-viable : 0 is put in the appropriate cell of the array $vTmp$. This treatment represents the value function updating in dynamic programming. In this case, a change is made, then $CheckChange$ is incremented. At the end of the step, the value function is transferred of $vTmp$ to v and we check $CheckChange$. If it is equal to 0, the algorithm can stop. It happens after a sufficient number of iterations and we then get the approximation of the viability kernel.

4 Results

First, we test the GPU programming on a simple example of population dynamics, in order to compare the performance with the sequential algorithm. Then, we use our parallel algorithm to treat a problem of bycatch fishery management up to 6 dimensions.

4.1 A multi-dimensional population model

One toy model is used here to evaluate the benefits of GPU programming. We consider the population model studied by Malthus and Verhulst. Malthusian ideas are the basis of the classical theory of population and growth (Ehrlich and Lui, 1997). This model is a simple dynamical system of population growth on a bounded space without any predator. The system state includes two variables : the size of the population and its evolution rate. Aubin and Saint-Pierre (2007) introduced a control on this evolution rate and adapted it to the management of renewable resources.

4.1.1 The model

As we want to analyze the performances of the parallelized algorithm on multi-dimensional problems, we artificially increase dimensions of the system in adding some independent variables in order to study the behaviour of the program in different dimensions. The new model is as follows : we considered n states x_i representing the size of n populations, which all have the same evolution rate $y(t) \in [d, e]$. The size of the populations must remain in $K = [a, b]$. The control is applied on the derivative of the evolution rate at each time step with the inertia bound c. The equations ruling the system in discrete time are :

$$x_i(t+1) = x_i(t) + x_i(t)\, y(t), \qquad i \in \{1, .., n\} \tag{7}$$

$$y(t+1) = y(t) + u(t) \tag{8}$$

with $-c \leq u(t) \leq +c$. The set $K = [a, b]^n \times [d, e]$ is the viability constraint set. For the rest of this work, bounds for K are set as $a = 0.2$, $b = 3$, $c = 0.5$, $d = -2$, $e = 2$. By default, the control is discretized in 5 values. Since the parallelization simply breaks down processes that are already independent in the usual algorithm, the discretization does not impact the accuracy of the results.

The program is implemented in C++, using CUDA 5. The computer CPU is an Intel® Xeon(R) CPU E5-2687W 0 @ 3.10 GHz x 16 and the GPU card is a Tesla K20C with a computing capability of 3.5 and 2496 CUDA cores.

4.1.2 Speed-up

Here we show a comparison between the two versions of the program applied to this model. The speed-up S is a measure evaluating the performance of the parallelization:

$$S = \frac{T_{seq}}{T_{par}} \tag{9}$$

where T_{seq} is the sequential algorithm execution time and T_{par} the execution time of the parallel one. The first one uses the usual algorithm; the second one uses the adapted algorithm to GPU computing.

We figure the evolution of the speed-up, in function of the size of the grid. In Fig. 5, we study the influence of grid refinement in the case of a 4-dimension problem.

First, according to Fig. 5a, we can see that the speed-up exceeds 5 in most cases of 4-dimension problems. It rises, using the whole GPU capacity but, then, the curve stagnates. The speed-up reaches a threshold when the GPU is fully loaded and where no more time gain is possible. On Fig. 5b, a comparison of elapsed times during the CPU and GPU version of the algorithm is shown. We notice that when the refinement of the grid is high, the computation time increases significantly in the CPU version.

Fig. 6, shows speed-up isolines in function of the number of dimensions and the number of points by dimension. We observe that when we have a 2- or 3-dimension problem, the speed-up is between 0 and 4. This is not significantly time-saving, and the initialization of the GPU can be slower than the classic array filling. However, when the number of dimensions grows, the speed-up increases up until 20 in 7 dimensions or more in 5 or 6 dimensions with a finer discretization. The speed-up isolevels follow the total number of points of the grid for high-dimension problems (4 dimensions or more). In these problems,

the main part of the time is spent in the trajectories storing part (the iterating kernel approximation ends quickly because of the low accuracy of the grid).

4.1.3 Parallelized parts

Here, we compare the speed-up during the storing of successors and during the computation one between the CPU and the GPU versions of the algorithm. In the following figure (Fig. 7), we study two different sizes of problems (2 and 6 dimensions) and we return speed-ups in function of the number of points by dimension. Elapsed times during the two parallelized parts are also returned.

On Fig.7a, we see that the elapsed time during the iterating kernel approximation is now higher than the one for storing the successors for 2-dimensions problems (for problems with more than 3000 points per dimension for the CPU version, or 1600 points per dimension for the GPU version). For problems in 3 dimensions or more (like 6 dimensions in Fig7c), storing successors time is still higher than iterating kernel approximation time. The time for storing successors exponentially increases with the number of dimensions and parallelizing this part of the algorithm is very efficient.

The speed-ups of the storing of trajectories and iterating kernel approximation are calculated separately here. As shown on Fig.7b and Fig.7d the speed-up increases up to 27 in 2 dimensions and 34 in 6 dimensions for the storing of trajectories, and up to 15 in 2 dimensions and 3.7 in 6 dimensions for the iterating kernel approximation.

For a small problem with a low discretization, the time taken by the two algorithms is negligible, but we observe that the speed-up is below one (up to 400 points by dimension in the 2D-problem, and 11 points by dimension in the 6D-problem). The mandatory initialization of the CUDA device explains the less successful results of the GPU version in smaller problems.

Moreover, we notice that the parallelization of the storing of successors is more efficient whatever the number of dimensions. Actually, memory exchanges (reading the value function array in the global memory from each thread at each step) in the second part alter the performance of the parallelization. Thus, the speed-up of the iterating kernel approximation falls down to 3.7 in 6 dimensions. Since the time elapsed during this step is small compared to the time elapsed during the storing of trajectories, its impact is not significant on the global speed-up.

4.2 Application to a bycatch fishery

We illustrate our results on a multidimensional bycatch fishery model. This model is an extension of the one studied by Krawczyk et al (2013) from 3 dimensions to 6 dimensions. We study the influence of the number of bycatch species on the fishery management.

4.2.1 The two-species dynamics

The model of Krawczyk is reminded here. There are two ecologically independent populations of fish x and y, harvested by a single fleet. The fishery focuses on harvesting the target stock x. We define a catchability coefficient q_x determining the quantity of biomass that each unit of effort extracts, relative to the total size of the biomass at the time. Thus, the harvest rate at time t is:

$$h_x(t) = q_x e(t) x(t) \tag{10}$$

The product of q_x and $e(t)$ is generally termed the level of fishing mortality and it is expressed as a proportion of biomass.

A linear bycatch production function is used :

$$h_y(t) = \alpha h_x(t) \tag{11}$$

where the constant α, $0 < \alpha < 1$ is the bycatch to target harvest ratio.

The respective population dynamics are then described by :

$$x(t+1) = x(t) + r_x x(t)\left(1 - \frac{x(t)}{L_x}\right) - q_x e(t) x(t) \tag{12}$$

$$y(t+1) = y(t) + r_y y(t)\left(1 - \frac{y(t)}{L_y}\right) - \alpha q_x e(t) x(t) \tag{13}$$

where r_x , r_y, L_x and L_y are all positive constants. By convention, r_x and r_y represent the intrinsic growth rates, and L_x and L_y are the environment carrying capacities, of x and y, respectively. According to Krawczyk, the numerical values for the bycatch species are chosen to have a bycatch stock less productive and less valuable than the target stock.

We suppose that the effort e can be controlled:

$e(t+1) = e(t) + u(t)\,,\ u(t) \in [u_{min}, u_{max}]$

The speed at which the regulator can change fishing intensity is bounded by u_{min} and u_{max}. The fishing fleet's profit is given by :

$$\pi_{xy}(t) = p_x h_x(t) + p_y h_y(t) - ce(t) - C \tag{14}$$

where p_y represents the unit prices of y, p_x the unit price of x The marginal cost of the fishing effort is c and C is a fixed cost. We define the viability constraints set K :

$$K = \left\{ (x, y, e, u) : x(t) \geq \frac{L_x}{10},\, y(t) \geq \frac{L_y}{10},\, \pi_{xy}(t) \geq 0,\, e(t) \in [e_{min}, e_{max}] \right\} \tag{15}$$

The safe minimum biomass levels are set to one tenth of each fish population's unexploited level, as commonly implemented in fisheries worldwide (Krawczyk et al, 2013).

4.2.2 The n-species dynamics

We propose to extend this model to treat n-species dynamics in order to illustrate the benefits of the GPU computation of the viability kernel. The dynamics of the target species remains unchanged (Eq.2). We suppose here that we have n bycatch species y_i. Moreover, we assume that these species are at the same trophic level, competitively sharing the same food resource (Ekerhovd and Steinshamm, 2014; Rice et al, 2013). The bycatch species growth is limited by the common carrying capacity L_y. The discrete dynamics becomes:

$$y_i(t+1) = y_i(t) + r_{y_i} y_i(t) \left(1 - \frac{\sum_{k=1}^{n-1} y_k(t)}{L_y} \right) - \alpha_i q_x e(t)\, x(t)\,,\ \forall i \in [1, n-1] \tag{16}$$

r_{y_i} is the intrinsic growth rate. α_i denotes the bycatch to target harvest ratio, a measure of how highly coupled the production relationship is. Each bycatch stock y_i is a by-product of the production process. A linear bycatch production function is used :

$$h_{y_i}(t) = \alpha_i h_x(t) \tag{17}$$

We assume here that the sum of all bycatch harvest ratios α_i is equal to α defined in the previous section :

$$\sum_{i=1}^{n-1} \alpha_i(t) = \alpha \tag{18}$$

For reasons of simplicity, we suppose that all the α_i are equal :

$$\alpha_i = \frac{\alpha}{n} \tag{19}$$

As before, economic sustainability requires that the activity remains profitable. The fishing fleet's profit is given by :

$$\pi_{xy}(t) = p_x h_x(t) + \sum_{i=1}^{n-1} p_{y_i} h_{y_i}(t) - ce(t) - C \tag{20}$$

Then, we have the viability constraint set K :

$$K = \left\{ (x, y_1, ..., y_{n-1}, e, u) : x(t) \geq \frac{L_x}{10}, \forall i \in [1, n-1], y_i(t) \geq \frac{L_y}{10n}, \pi_{xy}(t) \geq 0, e(t) \in [e_{min}, e_{max}] \right\} \tag{21}$$

In the following, we study dynamics with 1, 2, 3 or 4 bycatch species, which means dynamics with 3, 4, 5 or 6 dimensions respectively (including the fishing effort and the target species biomass).

4.2.3 Results

We use the parameters calibrated by Krawczyk (Krawczyk et al, 2013) and summed up in the Table 1.

With only one bycatch species, a similar shape of the viability kernel is obtained than the one in the previous study (Krawczyk et al, 2013). This

viability kernel is presented in Fig.8. For the following figures, the biomasses are scaled between 0.1 and 1.

The bounds of the figure are the constraint domain K. From each state contained inside the viability kernel, it exists at least a control strategy keeping the fishery in K. The states outside of the viability kernel are not viable, there is no control strategy satisfying the bio-economic constraints. For these states, the fishery is facing a "crisis" situation.

Now, bycatch species are added to the environment. For one bycatch species, 41.9% of states are inside the viability kernel. It decreases until less than 7% for four bycatch species.

The Fig.9 shows 3D slices of the 4D kernel for different fishing effort values. It represents the biomass of the target species and the biomass of the bycatch species required to be viable to satisfy the ecological constraints while maintaining the profitability of the fishery. A small fishing effort needs big available stock of the target biomass in order to conserve the system profitability. A mean fishing effort leads to a large slice of the viability kernel according to the target biomass. Finally, keeping a big fishing effort requires a mean target species stock to not deplete the bycatch populations. The results are displayed for 4D problems, but they can be extended to higher dimension problems.

Finally, we show on Fig.10 the number of states belonging to the viability kernel for a 6D problem with a grid of 20 points by dimension. The biggest viability kernel is obtained when the starting population of bycatch species are equal to $\frac{L_y}{n}$ (the offset is set by the grid step which is nearly 15.3 here). The environment is then completely filled with balanced increasing the possibility of sustainability. For larger initial populations, the environment is overloaded, causing the depletion of bycatch species. When the initial populations are

smaller, the ecological sustainability requirement restricts the number of viable states.

5 Discussion and conclusion

The GPU parallelization provides significantly faster viability kernel computations and tackles problems with a higher number of dimensions or more precise discretization than a sequential algorithm.

However, a huge storage space is required to save all the coordinates and all the successors for high-dimension dynamical problems. For example, in C++, storing one coordinate in double-precision floating-point format required 8 bytes (64 bits). Then, storing the coordinates of 100 millions points in a 8-dimension problem (10 points per dimension) requires 6.4Gb. Moreover the successors multiplied this value by the number of possible values of the control. Because of this size, it is impossible to send all the data at once. Instead, we cut the problem in smaller parts : computing the coordinates for some grid points and then computing the successors $g(x, u)$ for these points, the result is then saved. The process then continues with other grid points until all the whole grid has been processed. This reduces the need for memory.

This parallelized version may be improved with other tools (like STXXL in C++ to use large arrays) and using the mapped memory, which is based on some pointers to the RAM. Further improvements could be achieved by associating multi-CPUs and multi-GPUs architectures, the first ones cutting the problem in small parts and the second ones solving the subproblems.

The parallelization approach proposed here opens up perspectives in terms of parallelization for other viability problems. Then, revisiting other viability approaches such as an extension to stochastic dynamics (Doyen and De Lara, 2010) or algorithms such as support vector machine algorithm (Deffuant

et al, 2007) could also lead to significant improvements of efficiency. Finally, the parallelization of the viability kernel algorithm provides a tool for the management of socio-ecosystems as illustrated with the management of the multi-species bycatch fishery.

Acknowledgments

This work was supported by grants from Irstea and Région Auvergne. This support is gratefully acknowledged.

References

Andrés-Domenech P, Saint-Pierre P, Smala Fanokoa P, Zaccour G (2014) Sustainability of the Dry Forest in Androy: A Viability Analysis. Ecological Economics 104:33–49, DOI 10.1016/j.ecolecon.2014.04.016, URL http://linkinghub.elsevier.com/retrieve/pii/S0921800914001256

Aubin JP, Saint-Pierre P (2007) An Introduction to Viability Theory and Management of Renewable Resources. In: Advanced Methods for Decision Making and Risk Management in Sustainability Science, Nova Science Publishers, Inc, pp 43–93

Béné C, Doyen L, Gabay D (2001) A viability analysis for a bio-economic model. Ecological Economics 36(3):385–396, DOI 10.1016/S0921-8009(00)00261-5, URL http://linkinghub.elsevier.com/retrieve/pii/S0921800900002615

Bernard C, Martin S (2013) Comparing the sustainability of different action policy possibilities: application to the issue of both household survival and forest preservation in the corridor of Fianarantsoa. Mathematical biosciences 245(2):322–30, DOI 10.1016/j.mbs.2013.08.002, URL http://www.ncbi.nlm.nih.gov/pubmed/23954403

Bokanowski O, Martin S, Munos R, Zidani H (2006) An anti-diffusive scheme for viability problems. Applied Numerical Mathematics 56(9):1147–1162, DOI 10.1016/j.apnum.2006.03.004, URL http://linkinghub.elsevier.com/retrieve/pii/S0168927406000419

Bonneuil N (2006) Computing the viability kernel in large state dimension. Journal of Mathematical Analysis and Applications 323(2):1444–1454, DOI 10.1016/j.jmaa.2005.11.076, URL http://linkinghub.elsevier.com/retrieve/pii/S0022247X05012916

Cekmez U, Ozsiginan M, Technologies S, Air T, Academy F, Sahingoz OK (2013) Adapting the GA Approach to Solve Traveling Salesman Problems on CUDA Architecture. In: 14th IEEE International Symposium on Computational Intelligence and Informatics, pp 423–428

Chapel L, Deffuant G, Martin S, Mullon C (2008) Defining yield policies in a viability approach. Ecological Modelling 212(1-2):10–15, DOI 10.1016/j.ecolmodel.2007.10.007, URL http://linkinghub.elsevier.com/retrieve/pii/S030438000700525X

Chapel L, Castelló X, Bernard C, Deffuant G, Eguíluz VM, Martin S, San Miguel M (2010) Viability and resilience of languages in competition. PLoS ONE 5(1):1–11, DOI 10.1371/journal.pone.0008681

Deffuant G, Chapel L, Martin S (2007) Approximating Viability Kernels With Support Vector Machines. IEEE Transactions on Automatic Control 52(5):933–937, DOI 10.1109/TAC.2007.895881, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4200855

Djeridane B, Lygeros J (2008) Approximate Viability using Quasi-Random Samples and a Neural Network Classifier. In: The International Federation of Automatic Control, 2, pp 14,342–14,347

Donoho DL (2000) High-Dimensional Data Analysis: The Curses and Blessings of Dimensionality. In: Math Challenges of the 21st Century, pp 1–32

503 Doyen L, De Lara M (2010) Stochastic viability and dynamic pro-
504 gramming. Systems & Control Letters 59(10):629–634, DOI 10.1016/j.
505 sysconle.2010.07.008, URL http://linkinghub.elsevier.com/retrieve/
506 pii/S0167691110000952
507 Doyen L, Thébaud O, Béné C, Martinet V, Gourguet S, Bertignac M, Fi-
508 fas S, Blanchard F (2012) A stochastic viability approach to ecosystem-
509 based fisheries management. Ecological Economics 75:32–42, DOI 10.1016/j.
510 ecolecon.2012.01.005, URL http://linkinghub.elsevier.com/retrieve/
511 pii/S0921800912000067
512 Ehrlich I, Lui F (1997) The problem of population and growth: a review of the
513 literature from Malthus to contemporary models of endogenous population
514 and endogenous growth. Journal of economic dynamics & control 21(1):205–
515 42, URL http://www.ncbi.nlm.nih.gov/pubmed/12292267
516 Ekerhovd NA, Steinshamm SI (2014) Optimization in the 'Pelagic Complex':
517 A Multi-Species Competition Model of North East Atlantic Fisheries
518 Goldsworthy MJ (2014) A GPU–CUDA based direct simulation Monte Carlo
519 algorithm for real gas flows. Computers & Fluids 94:58–68, DOI 10.
520 1016/j.compfluid.2014.01.033, URL http://linkinghub.elsevier.com/
521 retrieve/pii/S0045793014000516
522 Kaynama S, Oishi M (2013) A Modified Riccati Transformation for Decen-
523 tralized Computation of the Viability Kernel Under LTI Dynamics. IEEE
524 Transactions on Automatic Control 58(11):2878–2892, arXiv:1302.5990v1
525 Krawczyk JB, Pharo A, Serea OS, Sinclair S (2013) Computation of via-
526 bility kernels: a case study of by-catch fisheries. Computational Manage-
527 ment Science 10(4):365–396, DOI 10.1007/s10287-013-0189-z, URL http:
528 //link.springer.com/10.1007/s10287-013-0189-z
529 Langdon WB (2011) Debugging CUDA. Proceedings of the 13th annual con-
530 ference companion on Genetic and evolutionary computation - GECCO

'11 p 415, DOI 10.1145/2001858.2002028, URL http://portal.acm.org/citation.cfm?doid=2001858.2002028

Maidens JN, Kaynama S, Mitchell IM, Oishi MMK, Dumont GA (2009) Lagrangian methods for approximating the viability kernel in high-dimensional systems. Automatica (2013) 49(7):2017–2029

Mametjanov A, Lowell D, Ma CC, Norris B (2012) Autotuning Stencil-Based Computations on GPUs. 2012 IEEE International Conference on Cluster Computing pp 266–274, DOI 10.1109/CLUSTER.2012.46, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6337788

Mathias JD, Bonté B, Cordonnier T, de Morogues F (2015) Using the Viability Theory to Assess the Flexibility of Forest Managers Under Ecological Intensification. Environmental Management (JUNE), DOI 10.1007/s00267-015-0555-4, URL http://link.springer.com/10.1007/s00267-015-0555-4

Mattioli J, Artiouchine K (2003) Noyau de viabilité : une contrainte globale pour la modélisation de systèmes dynamiques. Technique et Science Informatiques pp 19–32

Rice J, Daan N, Gislason H, Pope J (2013) Does functional redundancy stabilized fish communities 70:734–742

Rougé C, Mathias JD, Deffuant G (2013) Extending the viability theory framework of resilience to uncertain dynamics, and application to lake eutrophication. Ecological Indicators 29:420–433, DOI 10.1016/j.ecolind.2012.12.032, URL http://linkinghub.elsevier.com/retrieve/pii/S1470160X13000204

Sabo D, Barzelay O, Weiss S, Furst M (2014) Fast evaluation of a time-domain non-linear cochlear model on GPUs. Journal of Computational Physics 265:97–112, DOI 10.1016/j.jcp.2014.01.044, URL http://linkinghub.elsevier.com/retrieve/pii/S0021999114000898

559 Saint-Pierre P (1990) Approximation of the Viability Kernel. Applied Mathe-
560 matics & Optimization 29(2):1–22
561 Sicard M, Perrot N, Reuillon R, Mesmoudi S, Alvarez I, Martin S
562 (2012) A viability approach to control food processes: Application to a
563 Camembert cheese ripening process. Food Control 23(2):312–319, DOI
564 10.1016/j.foodcont.2011.07.007, URL http://linkinghub.elsevier.com/
565 retrieve/pii/S0956713511002805
566 Turriff J, Broucke ME (2009) A method to construct viability kernels for
567 nonlinear control systems. 2009 American Control Conference 1(1):3983–
568 3988, DOI 10.1109/ACC.2009.5160409, URL http://ieeexplore.ieee.
569 org/lpdocs/epic03/wrapper.htm?arnumber=5160409

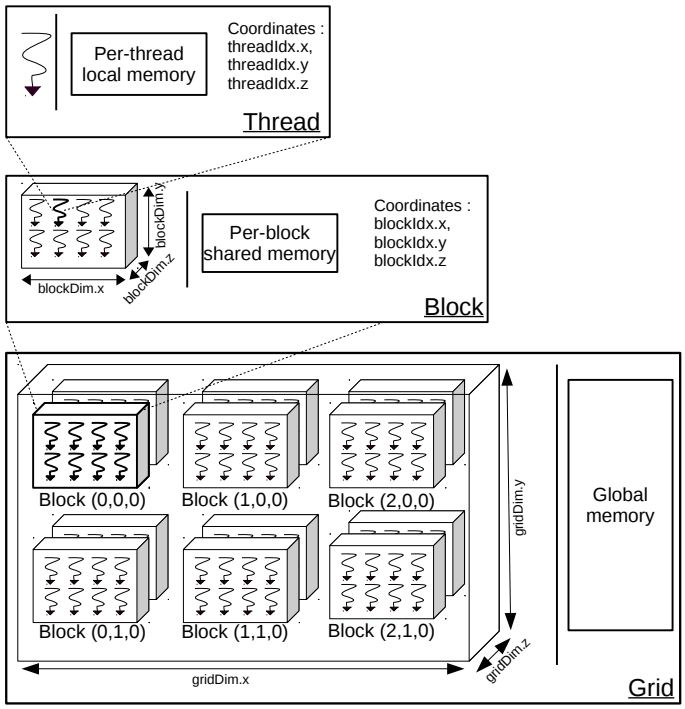


Fig. 1 GPU device architecture. The device is composed of a grid of threads blocks. Each level posseses its own memory. The parallelized tasks, like computation of controlled evolutions and computations of the value function, are made separately on the threads. A grid of 3x2x2 blocks of 4x2x1 threads is shown.

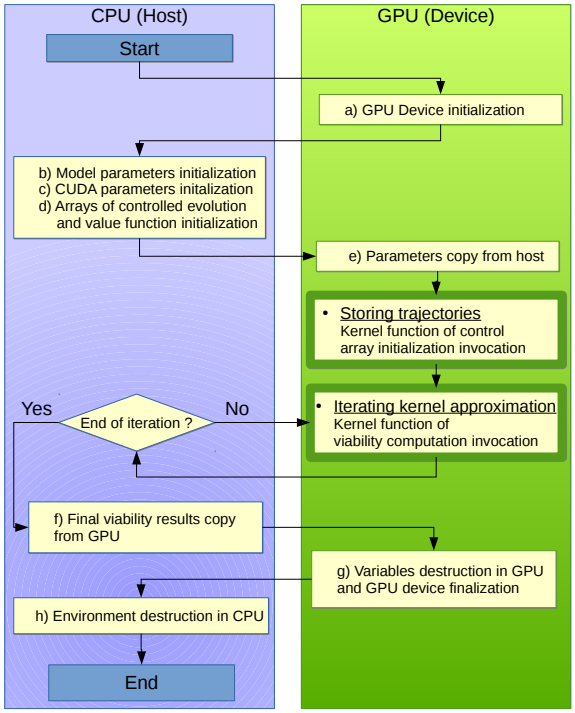


Fig. 2 Organization of the algorithm. Steps a,b,c,d,e,f,g and h are explained in 3.2. Kernel functions are detailed in 3.3 and in 3.4.

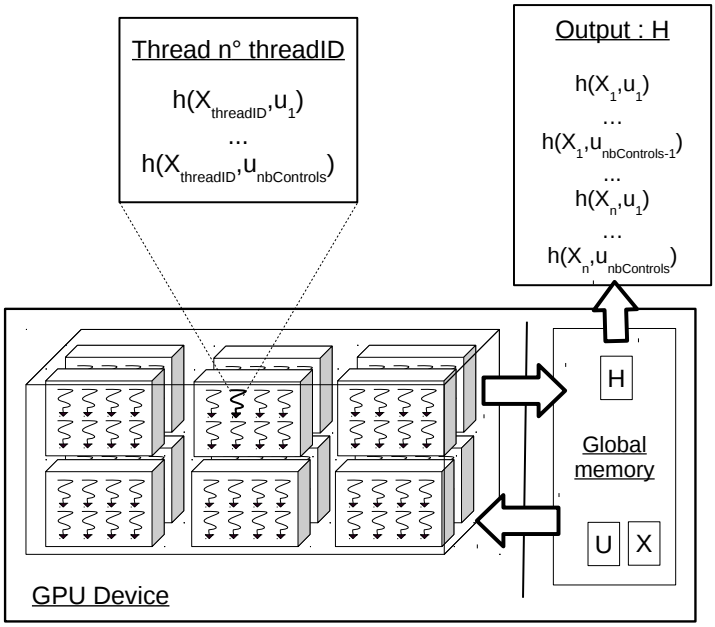


Fig. 3 Storing of trajectories : the parallelized controlled evolution computation from coordinates. The different steps of the parallelization are shown. The computation of all the controlled evolution for one point is done on one thread.

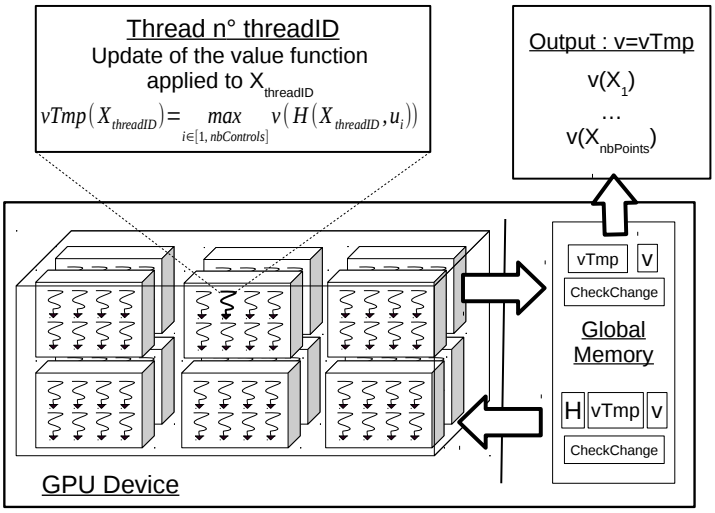
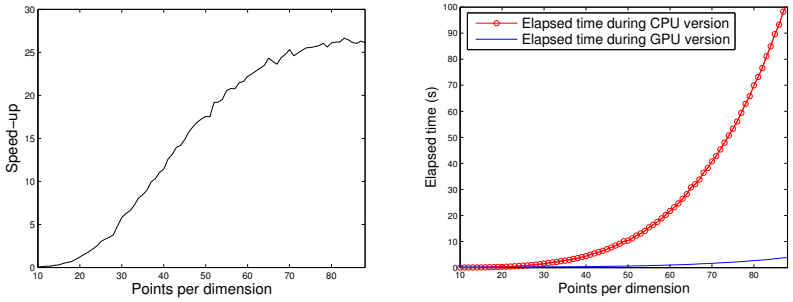


Fig. 4 Iterating kernel approximation: the parallelized viability computation. The different steps of the parallelization are shown.



(a) Speed-up (b) Comparison of elapsed times

Fig. 5 Effect of grid refinement on the speed-up in a 4-dimensions problem. The elapsed time during the two versions of the algorithm are shown in (b).

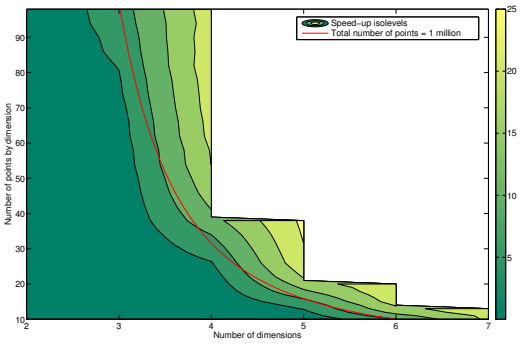
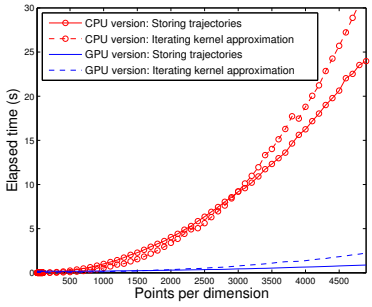
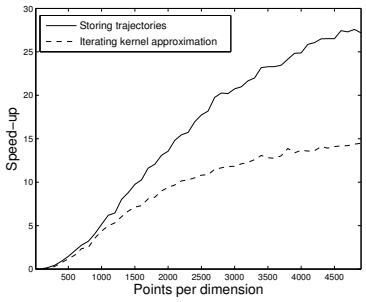


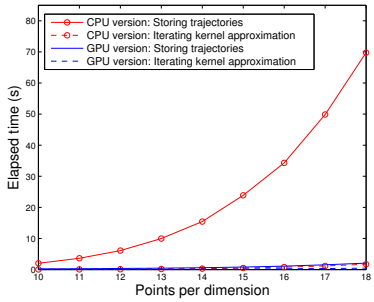
Fig. 6 Speed-up isolevels. The red line shows problems with same amount of points on the grid (1 million).



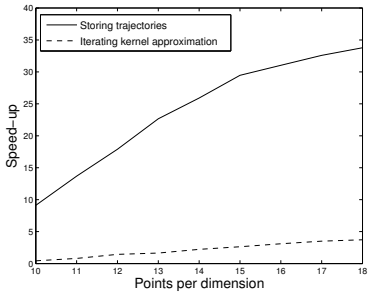
(a) Comparison of elapsed times (2D)



(b) Speed-up of the storing of trajectories and iterating kernel approximation (2D)



(c) Comparison of elapsed times (6D)



(d) Speed-up of the storing of trajectories and iterating kernel approximation (6D)

Fig. 7 Comparison of the storing of trajectories and the iterating kernel approximation elapsed times and speed-up between the GPU and CPU versions of the algorithm. Two sizes of problems are shown : 2-dimension problem (a) and (b), and 6-dimension problem (c) and (d).

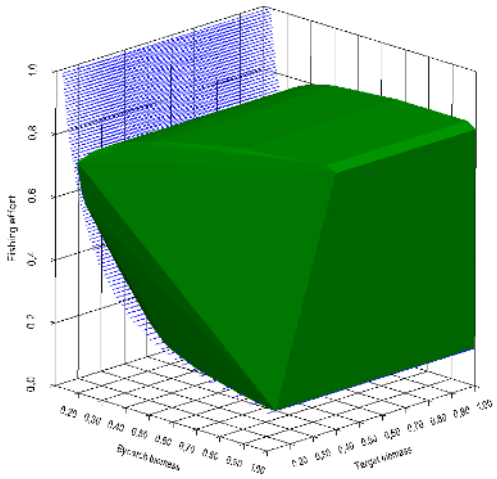
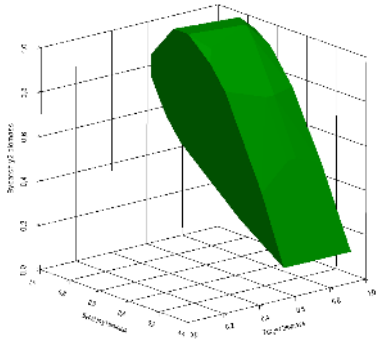
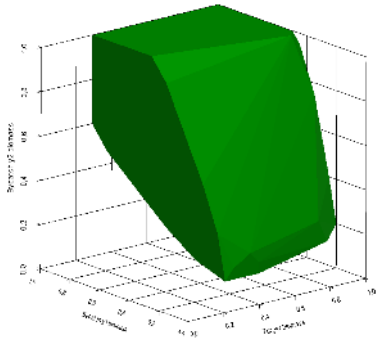


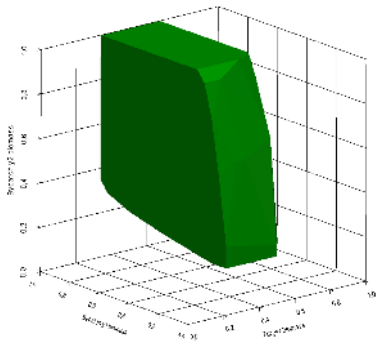
Fig. 8 Viability kernel for the 3D fishery model (with one bycatch species).Viability kernel for the 3D fishery model (with one bycatch species). The blue surface symbolizes the zero-profit surface. Having a positive profit (being above the blue surface) is a property of the viability constraint set K.



(a) Slice through $e = 0.2$



(b) Slice through $e = 0.5$



(c) Slice through $e = 0.8$

Fig. 9 3D slices of the 4D kernel for different initial values of the fishing effort e.

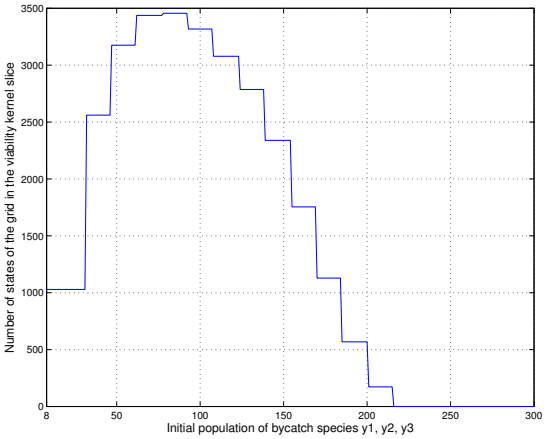


Fig. 10 Number of states of the grid belonging to the viability kernel for a 6D problem. The starting populations of three bycatch species are equal.

Name	Variable	Value
Target carrying capacity	L_x	600
Target growth rate	r_x	0.4
Target unit price	p_x	4
Target catchability coefficient	q_x	0.5
Bycatch common carrying capacity	L_y	300
Bycatch growth rate	r_{y_i}	0.2
Bycatch unit price	p_{y_i}	1.9
Bycatch harvest ratio	α_{y_i}	0.2
Marginal cost	c	10
Fixed cost	C	150
Maximum effort	e_{max}	1
Minimum effort	e_{min}	0.1
Maximum effort variation	u_{max}	0.01
Minimum effort variation	u_{min}	-0.01

Table 1 Parameters of the bycatch fishery. The parameters for all the bycatch species i are the same.