TI 2013-055/III
Tinbergen Institute Discussion Paper

Parallel Sequential Monte Carlo for Efficient Density Combination: The Deco Matlab Toolbox

Roberto Casarin[1]
Stefano Grassi[2]
Francesco Ravazzolo[3]
Herman K. van Dijk[4]

[1] *University Ca' Foscari of Venice and GRETA;*
[2] *CREATES, Aarhus University;*
[3] *Norges Bank, and BI Norwegian Business School;*
[4] *Erasmus University Rotterdam, and VU University Amsterdam.*

PARALLEL SEQUENTIAL MONTE CARLO FOR EFFICIENT DENSITY COMBINATION: THE DECO MATLAB TOOLBOX

ROBERTO CASARIN†, STEFANO GRASSI‡, FRANCESCO RAVAZZOLO♯, HERMAN K. VAN DIJK||

† *University Ca' Foscari of Venice and GRETA*
‡ *CREATES, Department of Economics and Business, Aarhus University*
♯ *Norges Bank and BI Norwegian Business School*
|| *Erasmus University Rotterdam, VU University Amsterdam and Tinbergen Institute*

ABSTRACT. This paper presents the Matlab package DeCo (Density Combination) which is based on the paper by Billio et al. (2013) where a constructive Bayesian approach is presented for combining predictive densities originating from different models or other sources of information. The combination weights are time-varying and may depend on past predictive forecasting performances and other learning mechanisms. The core algorithm is the function DeCo which applies banks of parallel Sequential Monte Carlo algorithms to filter the time-varying combination weights. The DeCo procedure has been implemented both for standard CPU computing and for Graphical Process Unit (GPU) parallel computing. For the GPU implementation we use the Matlab parallel computing toolbox and show how to use General Purposes GPU computing almost effortless. This GPU implementation comes with a speed up of the execution time up to seventy times compared to a standard CPU Matlab implementation on a multicore CPU. We show the use of the package and the computational gain of the GPU version, through some simulation experiments and empirical applications.

JEL codes: C11, C15, C53, E37.
Keywords: Density Forecast Combination, Sequential Monte Carlo, Parallel Computing, GPU, Matlab.

Date: April 8, 2013.
Roberto Casarin's research is supported by the Italian Ministry of Education, University and Research (MIUR) PRIN 2010-11 grant, and by the European Commission Collaborative Project SYRTO. Stefano Grassi acknowledges support from CREATES - Center for Research in Econometric Analysis of Time Series (DNRF78), funded by the Danish National Research Foundation. The views expressed in this paper are our own and do not necessarily reflect those of Norges Bank.

1. Introduction

Combining forecasts from different statistical models or other sources of information is a crucial issue in many different fields of science. Several papers have been proposed to handle this issue with Bates and Granger (1969) as one of the first attempt in this field. Initially the focus was on defining and estimating combination weights for point forecasting. For instance, Granger and Ramanathan (1984) propose to combine forecasts with unrestricted least squares regression coefficients as weights. Terui and van Dijk (2002) generalize least squares weights by specifying the weights in the dynamic forecast combination as a state space model with time-varying weights that are assumed to follow a random walk process. Recently, research interest has shifted to the construction of combinations of predictive *densities* (and not point forecasts) as well as to allow for model set *incompleteness* (the true model may not be included in the set of models for prediction) and *learning*. Further, different *model evaluation* criteria are used. Hall and Mitchell (2007) and Geweke and Amisano (2010) propose to use combination schemes based on Kullback-Leibler score; Gneiting and Raftery (2007) recommend strictly proper scoring rules, such as the Cumulative Rank Probability Score, in particular, if the focus is on some particular area, such as extreme tails, of the distribution. Billio et al. (2013) (hereby BCRVD (2013)) provide a general Bayesian distributional state space representation of predictive densities and specify combination schemes that allow for an incomplete set of models and different learning mechanisms and scoring rules.

The design of algorithms for a numerically efficient combination remains a challenging issue (e.g., see Gneiting and Raftery, 2007). BCRVD (2013) propose a combination algorithm based on Sequential Monte Carlo filtering. The proposed algorithm makes use of a random grid from the set of predictive densities and runs a particle filter at each point of the grid. The procedure is computational intensive, when the number of models to combine increases. A contribution of this paper is to present a Matlab package DeCo (Density Combination) for the combination of predictive densities, and a simple GUI for the use of this package.

This paper provides, through the DeCo package, an efficient implementation of BCRVD (2013) algorithm based on CPU and GPU parallel computing. We make use of recent increases in computing power and recent advances in parallel programming techniques. The focus of the microprocessor industry, mainly driven by Intel and AMD, has shifted from maximizing the performance of a single core to integrating multiple cores in one chip, see Sutter (2005) and Sutter (2011). Contemporaneously, the needs of the video game industry, requiring increasing computational performance, boosted the development of the Graphics Processing Unit (GPU), which enabled massively parallel computation.

In the present paper, we follow the recent trend of using GPUs for general, non-graphics, applications (prominently featuring those in scientific computing) the so-called General-Purpose computing on Graphics Processing Unit (GPGPU). The GPGPU has been applied successfully in different fields such as astrophysics, biology, engineering, and finance, where quantitative analysts started using this technology well ahead use by academic economists, see Morozov and Mathur (2011) for a literature review.

To date, the adoption of GPU computing technology in economics and econometrics has been relatively slow compared to other fields. There are a few papers that deal with this interesting topic, see Morozov and Mathur (2011), Aldrich et al. (2011), Geweke and Durham (2012), and Dziubinski and Grassi (2013). This is odd given the fact that parallel computing in economics has a long history. An early attempt to use parallel computation for Monte Carlo simulation is Chong and Hendry (1986), while Swann (2002) develops parallel implementation of maximum likelihood estimation. Creel and Goffe (2008) discuss a number of economic and econometric problems where parallel computing can be applied. The low diffusion of this technology in economics and econometrics, according to Creel (2005), is mainly due to two issues, which are the high cost of the hardware as parallel CPU architectures and to the steep learning curve of dedicated programming languages as CUDA (Compute Unified Device Architecture), OpenCL, Thrust and C++ AMP. Table 1 compares different currently available GPGPU approaches. The recent increase of attention to parallel computing is motivated by the fact that the hardware costs issue has been solved by the introduction of modern GPUs with relatively low cost. Nevertheless, the second issue remains still open. For example, Lee et al. (2010) report that a programmer proficient in C, a programming skill that can take some times to be learned, should be able to code effectively in CUDA within a few weeks.

We aim to contribute to this stream of literature by showing that GPU computing can be carried out almost without any extra effort using the parallel toolbox of Matlab and a suitable approach to Matlab coding of the algorithms. The Matlab environment allows easy use of GPU programming without learning CUDA. We emphasize that this paper is not intended to compare CPU and GPU computing. In fact, we propose the combination algorithm for both standard parallel CPU and for parallel GPU computation. Our simulation and empirical exercises show that DeCo GPU version is faster than standard sequential CPU version up to 70 times.

The structure of the paper is as follows. Section 2 introduces the principles of density forecast combinations with time-varying weights and parallel Sequential Monte Carlo algorithms. Section 3 presents a parallel Sequential Monte Carlo algorithm for density combinations. It also provides background material on GPU computing in Matlab. Sections 4 and 5 present simulation comparisons between GPU and CPU. Section 6 reports the results for the macroeconomic empirical application. Section 7 concludes.

TABLE 1. Comparison of different currently available GPGPU approaches.

	Advantages	Disadvantages
CUDA	Free	Vendor Lock-in
OpenCL	Free Heterogeneous	Difficult to program
Thrust	Free Easy to program	Vendor Lock-in
C++ AMP	Open Standard Heterogeneous Free (Express Edition) Easy to program	Currently only Windows implementations exist

Appendix A describes the structure of the algorithm and Appendix B shows the package graphical interface.

2. TIME-VARYING COMBINATIONS OF PREDICTIVE DENSITIES

2.1. A combination scheme.

BCRVD (2013) introduces a general density combination scheme, which allows for time-varying weights; model set incompleteness (meaning the true model might not be in the model set); combination weight uncertainty and learning. The authors give a general distributional representation of the combination, provide an effective algorithm for the sequential estimation of the weights and discuss some alternative specifications of the combination and of weight dynamics. In the package, and in the simulation and empirical exercises presented in this paper, we apply for convenience the Gaussian combination scheme with logistic weights applied by BCRVD (2013).

Let $\mathbf{y}_t \in \mathcal{Y} \subset \mathbb{R}^L$ be the L-vector of observable variables at time t and $\tilde{\mathbf{y}}_t = (\tilde{\mathbf{y}}'_{1,t}, \dots, \tilde{\mathbf{y}}'_{K,t})' \in \mathcal{Y} \subset \mathbb{R}^{KL}$, with element $\tilde{\mathbf{y}}_{k,t} = (\tilde{y}^1_{k,t}, \dots, \tilde{y}^L_{k,t})' \in \mathcal{Y} \subset \mathbb{R}^L$ the typical one-step ahead predictor for $\mathbf{y}_t$ for the k-th model, $k = 1, \dots, K$, in the pool. The combination scheme is specified as:

$$p(\mathbf{y}_t|W_t, \tilde{\mathbf{y}}_t) \propto |\Sigma|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{y}_t - W_t\tilde{\mathbf{y}}_t)'\,\Sigma^{-1}\,(\mathbf{y}_t - W_t\tilde{\mathbf{y}}_t)\right\}$$

$t = 1, \dots, T$, where $W_t = (\mathbf{w}^1_t, \dots, \mathbf{w}^L_t)'$ is a weight matrix, with $\mathbf{w}^l_t = (\mathbf{w}^l_{1,t}, \dots, \mathbf{w}^l_{KL,t})'$ as the l-th row vector containing the combination weights for the KL elements of $\tilde{\mathbf{y}}_t$ and for the prediction of $y_{l,t}$.

The dynamics of the combination weights $w^l_{h,t}$, $h = 1, \dots, KL$ is

$$w^l_{h,t} = \frac{\exp\{x^l_{h,t}\}}{\sum_{j=1}^{KL} \exp\{x^l_{j,t}\}}, \quad \text{with } h = 1, \dots, KL$$

where

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}) \propto |\Lambda|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{x}_t - \mathbf{x}_{t-1})'\,\Lambda^{-1}\,(\mathbf{x}_t - \mathbf{x}_{t-1})\right\}$$

with $\mathbf{x}_t = \text{vec}(X_t) \in \mathbb{R}^{KL^2}$ where $X_t = (\mathbf{x}_t^1, \ldots, \mathbf{x}_t^L)'$. A learning mechanism can also be added to the weight dynamics, resulting in:

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{y}_{t-\tau:t-1}, \tilde{\mathbf{y}}_{t-\tau:t-1}) \propto |\Lambda|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{x}_t - \boldsymbol{\mu}_t)' \Lambda^{-1} (\mathbf{x}_t - \boldsymbol{\mu}_t)\right\}$$

where $\boldsymbol{\mu}_t = \mathbf{x}_{t-1} - \Delta \mathbf{e}_t$ and

$$e^l_{K(l-1)+k,t} = (1-\lambda)\sum_{i=1}^{\tau} \lambda^{i-1} f\left(y^l_{t-i}, \tilde{y}^l_{k,t-i}\right),$$

$k = 1, \ldots, K$, $l = 1, \ldots, L$, with λ a discount factor and τ the number of previous observations used in the learning. We assume $f()$ is an exponentially weighted learning strategy. Note that DeCo package relies on a general algorithm which can account for different scoring rules, such as the Kullback-Leibler score (Hall and Mitchell (2007) and Geweke and Amisano (2010)) and the Cumulative Rank Probability Score (Gneiting and Raftery (2007)).

The proposed state space representation of the combination scheme provides a forecast density for the observable variables, conditional on the predictors and on the combination weights. Moreover, the representation is quite general, allowing for nonlinear and non-Gaussian combination models. We use Sequential Monte Carlo algorithms, also known as Particle Filters, to estimate sequentially over time the optimal combination weights and the predictive density.

The steps of the density combination algorithms are sketched in the rest of this section. Let us denote with $\mathbf{v}_{1:t} = (\mathbf{v}_1, \ldots, \mathbf{v}_t)$ a collection of vectors $\mathbf{v}_t$ from $1, \ldots, t$. Let $\mathbf{w}_t = \text{vec}(W_t)$ be the vector of model weights associated with $\tilde{\mathbf{y}}_t$ and $\boldsymbol{\theta} \in \Theta$ the parameter vector of the combination model. Define the augmented state vector $\mathbf{z}_t = (\mathbf{w}_t, \boldsymbol{\theta}_t) \in \mathcal{Z}$ and the augmented state space $\mathcal{Z} = \mathcal{X} \times \Theta$ where $\boldsymbol{\theta}_t = \boldsymbol{\theta}$, $\forall t$. The distributional state space form of the forecast model is

$$\begin{array}{llll}
(1) & \mathbf{y}_t \sim & p(\mathbf{y}_t|\mathbf{z}_t, \tilde{\mathbf{y}}_t) & \text{(measurement density)} \\
(2) & \mathbf{z}_t \sim & p(\mathbf{z}_t|\mathbf{z}_{t-1}, \mathbf{y}_{1:t-1}, \tilde{\mathbf{y}}_{1:t-1}) & \text{(transition density)} \\
(3) & \mathbf{z}_0 \sim & p(\mathbf{z}_0) & \text{(initial density)}
\end{array}$$

The state predictive and filtering densities conditional on the predictive variables $\tilde{\mathbf{y}}_{1:t}$ are

$$(4) \quad p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}) = \int_{\mathcal{Z}} p(\mathbf{z}_{t+1}|\mathbf{z}_t, \mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}) p(\mathbf{z}_t|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}) d\mathbf{z}_t$$

$$(5) \quad p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t+1}, \tilde{\mathbf{y}}_{1:t+1}) = \frac{p(\mathbf{y}_{t+1}|\mathbf{z}_{t+1}, \tilde{\mathbf{y}}_{t+1}) p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t})}{p(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t})}$$

respectively, which represent the optimal nonlinear filter (see Doucet et al. (2001)). The marginal predictive density of the observable variables is then

$$p(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}) = \int_{\mathcal{Y}} p(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{t+1}) p(\tilde{\mathbf{y}}_{t+1}|\mathbf{y}_{1:t}) d\tilde{\mathbf{y}}_{t+1}$$

where $p(\mathbf{y}_{t+1}|\mathbf{y}_{1:t},\tilde{\mathbf{y}}_{t+1})$ is defined as

$$\int_{\mathcal{Z}\times\mathcal{Y}^t} p(\mathbf{y}_{t+1}|\mathbf{z}_{t+1},\tilde{\mathbf{y}}_{t+1})p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t},\tilde{\mathbf{y}}_{1:t})p(\tilde{\mathbf{y}}_{1:t}|\mathbf{y}_{1:t-1})d\mathbf{z}_{t+1}d\tilde{\mathbf{y}}_{1:t}$$

and represents the conditional predictive density of the observable given the past values of the observable and of the predictors.

2.2. **A combination algorithm.** The analytical solution of the optimal filter for non-linear state space models is generally not known. Approximate solutions are needed. We apply a numerical approximation method, that converges to the optimal filter in Hilbert metric, in the total variation norm and in a weaker distance suitable for random probability distributions (e.g., see Legland and Oudjane (2004)). More specifically we consider a sequential Monte Carlo (SMC) approach to filtering. See Doucet et al. (2001) for an introduction to SMC and Creal (2009) for a recent survey on SMC in economics. We propose to use banks of SMC filters, where each filter, is conditioned on a sequence of realizations of the predictor vector $\tilde{\mathbf{y}}_t$. The resulting algorithm for the sequential combination of densities is defined through the following steps.

Step 0. Initialize independent particle sets $\Xi_0^j = \{\mathbf{z}_0^{i,j},\omega_0^{i,j}\}_{i=1}^N$, $j=1,\ldots,M$. Each particle set Ξ_0^j contains N i.i.d. random variables $\mathbf{z}_0^{i,j}$ with random weights $\omega_0^{i,j}$. Initialize a random grid over the set of predictors, by generating i.i.d. samples $\tilde{\mathbf{y}}_1^j$, $j=1,\ldots,M$, from $p(\tilde{\mathbf{y}}_1|\mathbf{y}_0)$. We use the sample of observations $\mathbf{y}_0$ to initialize the individual predictors.

Step 1. At the iteration $t+1$ of the combination algorithm, we approximate the predictive density $p(\tilde{\mathbf{y}}_{t+1}|\mathbf{y}_{1:t})$ with the discrete probability

$$p_M(\tilde{\mathbf{y}}_{t+1}|\mathbf{y}_{1:t}) = \sum_{j=1}^{M}\delta_{\tilde{\mathbf{y}}_{t+1}^j}(\tilde{\mathbf{y}}_{t+1})$$

where $\tilde{\mathbf{y}}_{t+1}^j$, $j=1,\ldots,M$, are i.i.d. samples from the predictive densities and $\delta_x(y)$ denotes the Dirac mass centered at x. This approximation is also motivated by the forecasting practice (see Jore et al. (2010)). The predictions usually come, from different models or sources, in form of discrete densities. In some cases, this is the result of a collection of point forecasts from many subjects, such as surveys forecasts. In other cases the discrete predictive is a result of a Monte Carlo approximation of the predictive density (e.g. Importance Sampling or Markov-Chain Monte Carlo approximation of the model predictive density).

Step 2. We assume an independent sequence of particle sets $\Xi_t^j = \{\mathbf{z}_{1:t}^{i,j},\omega_t^{i,j}\}_{i=1}^N$, $j=1,\ldots,M$, is available at time $t+1$ and that each particle set provides the approximation

$$p_{N,j}(\mathbf{z}_t|\mathbf{y}_{1:t},\tilde{\mathbf{y}}^j_{1:t}) = \sum_{i=1}^{N} \omega_t^{i,j}\delta_{\mathbf{z}_t^{i,j}}(\mathbf{z}_t) \tag{6}$$

of the filtering density, $p(\mathbf{z}_t|\mathbf{y}_{1:t},\tilde{\mathbf{y}}^j_{1:t})$, conditional on the j-th predictor realization, $\tilde{\mathbf{y}}^j_{1:t}$. Then M independent SMC algorithms are used to find a new sequence of M particle sets, which include the information available from the new observation and the new predictors. Each SMC algorithm iterates, for $j = 1,\dots,M$, the following steps.

Step 2.a. The basic SMC algorithm uses the particle set to approximate the predictive density with an empirical density. More specifically, the predictive density of the combination weights and parameter, $\mathbf{z}_{t+1}$, conditional on $\tilde{\mathbf{y}}^j_{1:t}$ and $\mathbf{y}_{1:t}$ is approximated as follows

$$p_{N,j}(\mathbf{z}_{t+1}|\mathbf{y}_{1:t},\tilde{\mathbf{y}}^j_{1:t}) = \sum_{i=1}^{N} p(\mathbf{z}_{t+1}|\mathbf{z}_t,\mathbf{y}_{1:t},\tilde{\mathbf{y}}^j_{1:t})\omega_t^{i,j}\delta_{\mathbf{z}_t^{i,j}}(\mathbf{z}_t) \tag{7}$$

For the applications in the present paper we use a regularized version of the SMC procedure given above (e.g., see Liu and West (2001), Musso et al. (2001) and Casarin and Marin (2009)).

Step 2.b. We update the state predictive density by using the information coming from $\tilde{\mathbf{y}}^j_{t+1}$ and $\mathbf{y}_{t+1}$, that is

$$p_{N,j}(\mathbf{z}_{t+1}|\mathbf{y}_{1:t+1},\tilde{\mathbf{y}}^j_{1:t+1}) = \sum_{i=1}^{N} \gamma_{t+1}^{i,j}\delta_{\mathbf{z}_{t+1}^{i,j}}(\mathbf{z}_{t+1}) \tag{8}$$

where $\gamma_{t+1}^{i,j} \propto \omega_t^{i,j} p(\mathbf{y}_{t+1}|\mathbf{z}_{t+1}^{i,j},\tilde{\mathbf{y}}^j_{t+1})$ is a set of normalized weights.

Step 2.c. The hidden state predictive density can be used to approximate the observable predictive density as follows

$$p_{N,j}(\mathbf{y}_{t+1}|\mathbf{y}_{1:t},\tilde{\mathbf{y}}^j_{1:t+1}) = \sum_{i=1}^{N} \gamma_{t+1}^{i,j}\delta_{\mathbf{y}_{t+1}^{i,j}}(\mathbf{y}_{t+1}) \tag{9}$$

where $\mathbf{y}_{t+1}^{i,j}$ has been simulated from the combination model $p(\mathbf{y}_{t+1}|\mathbf{z}_{t+1}^{i,j},$ $\tilde{\mathbf{y}}^j_{t+1})$ independently for $i = 1,\dots,N$.

Step 2.d. The systematic resampling of the particles introduces extra Monte Carlo variations, see Liu and Chen (1998). This can be reduced be doing resampling only when the Effective Sample Size (ESS) is below a given threshold. The ESS is defined as

$$\mathrm{ESS}_t^j = \frac{N}{1 + N\sum_{i=1}^{N}\left(\gamma_{t+1}^{i,j} - N^{-1}\sum_{i=1}^{N}\gamma_{t+1}^{i,j}\right)^2 \Big/ \left(\sum_{i=1}^{N}\gamma_{t+1}^{i,j}\right)^2}\,.$$

and measures the overall efficiency of an importance sampling algorithm. At the $t+1$-th iteration if $\text{ESS}_{t+1}^{j} < \kappa$, simulate $\Xi_{t+1}^{j} = \{\mathbf{z}_{t+1}^{k_i,j}, \omega_{t+1}^{i,j}\}_{i=1}^{N}$ from $\{\mathbf{z}_{t+1}^{i,j}, \gamma_{t+1}^{i,j}\}_{i=1}^{N}$ (e.g., multinomial resampling) and set $\omega_{t+1}^{i,j} = 1/N$. We denote with k_i the index of the i-th re-sampled particle in the original set Ξ_{t+1}^{j}. If $\text{ESS}_{t+1}^{j} \geq \kappa$ set $\Xi_{t+1}^{j} = \{\mathbf{z}_{t+1}^{i,j}, \omega_{t+1}^{i,j}\}_{i=1}^{N}$.

Step 3. At the last step, obtain the following empirical predictive density

$$p_{M,N}(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}) = \frac{1}{MN}\sum_{j=1}^{M}\sum_{i=1}^{N}\omega_t^{i,j}\delta_{\mathbf{y}_{t+1}^{i,j}}(\mathbf{y}_{t+1}) \tag{10}$$

3. Parallel SMC for density combination: DeCo

Matlab is a popular software in the economics and econometrics community (e.g., see LeSage (1998)), which has recently introduced the support to GPU computing in its parallel computing toolbox. This allows to use raw CUDA code within a Matlab program as well as already build functions that are directly executed on the GPU. Using the build-in functions we show that GPGPU can be almost effortless where the only knowledge required is a decent programming skill in Matlab. With a little effort we provide GPU implementation of the methodology recently proposed by BCRVD (2013). This implementation comes with a speed up of the execution time up to hundred of times compared to a multicore CPU with a standard Matlab code.

3.1. GPU computing in Matlab.

There is little difference between the CPU and GPU Matlab code: Listings 1 and 2 report the same program which generates and inverts a matrix on CPU and GPU respectively.

```
iRows = 1000; iColumns = 1000;% Number of rows and
    columns
C_on_CPU =  randn(iRows, iColumns);% Generate Random
    number on the CPU
InvC_on_CPU = inv(C_on_CPU);% Invert the matrix
```

Listing 1. Matlab CPU code that generate and invert a matrix

```
iRows = 1000; iColumns = 1000;% Number of rows and
    columns
C_on_GPU =  gpuArray.randn(iRows, iColumns); % Generate
    Random number on the GPU
InvC_on_GPU = inv(C_on_GPU);% Invert the matrix
InvC_on_CPU = gather(InvC_on_GPU);% Transfer the data
    from the GPU to CPU
```

Listing 2. Matlab GPU code that generate and invert a matrix

The GPU code, Listing 2, uses the command *gpuArray.randn* to generate a matrix of normal random numbers. The build-in function *gpuArray.randn* is handled by the NVIDIA plug-in that generates the random number with an underline raw CUDA code. Once the variable *C_on_GPU* is created, standard functions such as *inv* recognize that the variable is on GPU memory and execute the corresponding GPU function, e.g. *inv* is executed directly on the GPU. This is completely transparent to the user. If further calculations are needed on the CPU then the command *gather(·)* transfer the data from GPU to the CPU, see line 3 of Listing 2. There exist already a lot of supported functions and this number continuously increases with new releases.

3.2. **Parallel Sequential Monte Carlo.** The structure of the GPU program, that is very similar to the CPU one, is reported in Appendix A. Before introducing our programming strategy we explain why the GPGPU computing has become very competitive for high parallel problems.

The GPU were created initially for $3D$ rendering, in other words, to create a $3D$ image on a monitor. A representation of a $3D$ scene in a monitor is composed of a set of points, known as vertices, that are based on $2D$ primitives, called triangles. To display this $3D$ scene, the GPU considers the set of all primitives as independent structures and computes various properties, such as lighting and visibility, independently one to another.

In graphical context the majority of these computations are executed in floating point, so GPUs were initially optimized for performing these types of computations. Lately, GPUs were extended to double precision calculation, see Section 4. Since a GPU performs a relatively small set of operations on a specific set of data points (each vertex on the screen), GPU makers (eg. NVIDIA and ATI) focus mainly on creating hardware that specializes in these tasks instead of a wide array of operations such as the CPU. This is a weak and a strong point at the same time, it restricts the set of problems in which the GPU can be used but allows to perform the specialized tasks more efficiently then the CPU.

The basic example of a high parallelizable problem is matrix multiplication and, in general, matrix linear algebra. These operations are very suitable for GPGPU computing because they can be easily divided into the large number of cores available on the GPU, see Gregory and Miller (2012) for an introduction.

At first sight our problem does not seem to be easily parallelizable. But a closer look shows that the only sequential part of the algorithm is the time iteration, indeed the results of time $t+1$ are dependent on t.

Our key idea is to rewrite in matrix form that part of the algorithm that iterates over particles and predictive draws in order to exploit the GPGPU computational efficiency. Following the notation in Section 2, we let M be the draws from the predictive densities, K the number of predictive models, L the number of variables to predict, T the sample size, and N the number of particles. Consider $L = 1$ for the sake of simplicity, then the code carries out a matrix of dimension $(MN \times K)$. The dimension could be large, e.g., in our simulation and empirical exercises they are $(5,000 \cdot 1,000 \times 3)$ and $(1,000 \cdot 1,000 \times 12)$ respectively. All the operations, such as addition and multiplication, become just matrix operations. GPU, as explained before, is explicitly designed to carry out these operations.

As an example of such a coding strategy, we describe the parallel version of the initialization step of the SMC algorithm (see first step of the diagram in Appendix A and subsection (Step 0) in Section 2). We apply a linear regression and then generate a set of normal random numbers to set the initial values of the states. Using a multivariate approach to the regression problem, we can perform it in just one single, big matrix multiplication and inversion for all draws. An example of initialization, similar to the one used in the package, is given in listing 3.[1]

```
 1 %% Built the block-diagonal input matrices yi1 and vY1
 2 yi1 = []; vY1 = [];
 3 for j=1:M
 4    yi1 = blkdiag(yi1, vYAll(:, :, j));
 5   % vYAll(:, :, j) has T rows and K columns
 6   vY1 = blkdiag(vY1, vY);
 7   % vY1 has T rows and 1 column
 8 end
 9 %% Load on the GPU
10 mX1GPU = gpuArray(yi1);
11 vY1GPU = gpuArray(vY1);
12 %% Initialize particles on the GPU
13 mOmega10 = inv(mX1GPU' * mX1GPU)* mX1GPU' * vY1GPU;
14 mMatrix = kron(gpuArray.ones(M, 1), gpuArray.eye(K));
15 mOmega10 = mOmega10' * mMatrix;
16 mOmega10 = kron(gpuArray.ones(N, 1), mOmega10) ...
17          + 5 * gpuArray.randn(N * M, K);
```

LISTING 3. Block Regression on the GPU

Listing 3 shows that the predictive densities and the observable variables are stacked in block-diagonal matrices (lines 1-8) of dimensions $(TM \times MK)$ and $(TM \times M)$ by using the command *blkdiag*$(\cdot)$, and then transferred to the GPU by the $gpuArray(\cdot)$ command (lines 10-11). The function $gpuArray.randn(\cdot)$ is then used to generate normal random numbers on the GPU and thus to initialize the value of the particles (lines 13-17).

This strategy is carried out all over the program and applied also to the simulation of the set of particles. For example, listing 4 reports a sample of the code for the SMC prediction step (see subsection (Step 2.a) in Section 2) for the latent states.

```
  mParticleTemp = S.omega1' + kron(gpuArray.ones(1, M * N),
      sqrt(Setting.Sigma)).* gpuArray.randn(K, M * N);
```

LISTING 4. Draws for the latent states

[1]In the package, we use the following labelling: *Setting.iDimOmega* for K, *Setting.iDraws* for M and *Setting.cN* for N.

The Kronecker product (fucntion $kron(\cdot)$) creates a suitable matrix of standard deviations. We notice that the matrix implementation of the filter requires availability of physical memory on the graphics card. If memory is not enough to run in parallel all the draws, then it is possible to split the M draws in $k = \frac{M}{m}$ blocks of size m and to run the combination algorithm sequentially over the blocks, and in parallel within the blocks.[2]

The only step of the algorithm which uses the CPU is resampling (see diagram in Appendix A and subsection (Step 2.d) in Section 2). The generated particles are copied to the CPU memory and after the necessary calculations, they are passed back to the GPU. This copying back and forth brings a computational time cost that can be high in small problems but becomes much less important as the number of particles and series increases.

4. Differences between CPU and GPU Monte Carlo

GPUs can execute calculations in single and double precision as defined by the IEEE 754 standard, IEEE (2008). Single precision numbers are only half the size of double precision and they are more limited in the range of values represented. If an application requires a high degree of precision, double precision numbers are the only possibility. We work with double precision numbers because our applications focus on density forecasting and precise estimates of statistical quantities, like extreme events that are in the tails of the predictive distribution, may be very important for economic and financial decisions.

The GPU card are very fast in single precision calculation but loose power in double precision. Therefore, some parameters should be set carefully to have a fair comparison between CPU and GPU. First, both programs are to be implemented in double precision. Second, the CPU program has to be parallel in order to use all available CPU cores. Third, the choice of the hardware is crucial, see Aldrich (2013) for a discussion. In all our exercises, we use a recent CPU processor, Intel Core i7-3820QM, launched in 2012Q2. This CPU has four physical cores that doubled thank to the HyperThreading technology. Not all users of DeCo might have access to such up-to-date hardware giving its costs. So we run CPU code also using a less expensive machine, the Intel Xeon X3430, launched in 2009Q3. To run the CPU code in parallel, Matlab requires the Parallel Computing Toolbox. We also investigate performance when such option is switched off and the CPU code is run sequentially.

The GPU used in this study is a NVIDIA Quadro K2000M. The card is available at a low cost, but also has low performance because it is designed for a mobile machine (as the suffix M means). A user with a desktop computer might have access to a more powerful video card, such as, e.g., a

[2]We run the blocks sequentially because Matlab has not yet a parallel for loop command for running in parallel the $k = \frac{M}{m}$ blocks of GPU computations. The DeCo parallel CPU version fixes $m = 1$ and parallelizes over the $k = M$ blocks.

NVIDIA GTX590 or a NVIDIA Tesla. We refer to Matlab for alternative cards and, in particular, to the function "GPUBench".

Finally we shall emphasize that the results of the CPU and GPU version of the same program are not necessarily the same. It is likely that the GPU results will be more precise. This is related to the so called fused multiply add (FMA) operations that GPUs support, see Whitehead and Fit-Florea (2011). The FMA operation carries along more accurate calculations, unfortunately there are currently no CPUs that support this new standard. To investigate the numerical difference between CPU and GPU, we provide some numerical integration exercises based on both standard Monte Carlo integration and Sequential Monte Carlo integration.

4.1. **Monte Carlo.** We consider six simple integration problems and compare their analytical solutions to their crude Monte Carlo (see Robert and Casella, 2004) numerical solutions. Let us consider the two integrals of the function f over the unit interval

$$\mu(f)=\int_0^1 f(x)dx, \quad \sigma^2(f)=\int_0^1 (f(x)-\mu(f))^2 dx.$$

The Monte Carlo approximations of the integrals are

$$\hat{\mu}_N(f)=\frac{1}{N}\sum_{i=1}^{N} f(X_i), \quad \hat{\sigma}^2_N(f)=\frac{1}{N}\sum_{i=1}^{N}(f(X_i)-\hat{\mu}_N(f))^2$$

where $X_1,\ldots,X_N$ is a sequence of N i.i.d. samples from a standard uniform distribution.

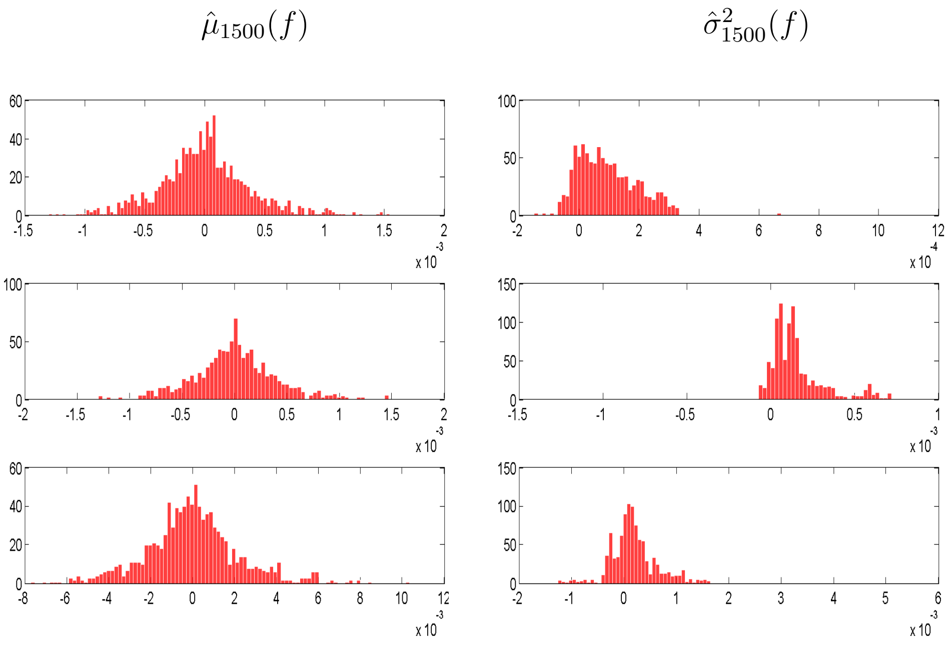
The numerical integration problems considered in the experiments correspond to the following choices of the integrand function:

(1) $f(x)=x$;
(2) $f(x)=x^2$;
(3) $f(x)=\cos(\pi x)$.

We repeat $G=1000$ times each Monte Carlo integration exercise with sample sizes $N=1500$. Finally, we compute the squared difference between the Monte Carlo and the analytical solution of the integral. The histogram of the differences between CPU and GPU results are given in Fig. 1. The more concentrated the density is on the negative part of the support, the higher is the precision of the CPU with respect to the GPU. Concentration on the positive part corresponds to a GPU overperfomance.

The histograms in the first column of Fig. 1 have positive standardized mean equal to 0.0051, 0.0059 and 0.0079 respectively and positive skewness equal to 0.3201, 0.2924 and 0.3394 respectively. Thus, it seems to us that GPU calculations are often more precise. From the histograms in the second column of Fig. 1, one can conclude that differences between GPU and CPU results are larger for the variance calculation. All histograms

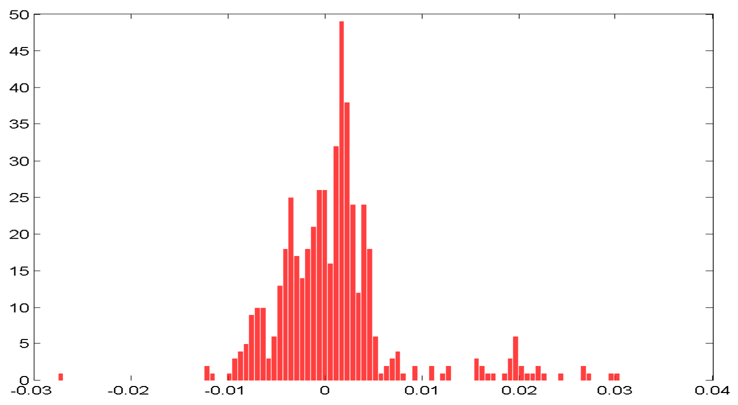
FIGURE 1. Histograms for the CPU-GPU mean square error differences in the MC estimators of $\mu_N(f)$ and $\sigma^2_N(f)$ (different columns), for different choices of f (different rows), using G replications.



have positive standardized mean equal to 0.9033, 0.8401 and 0.3843 respectively and positive skewness equal to 1.6437, 0.0771 and 1.8660 respectively. Nevertheless, we checked the statistical relevance of the differences between CPU and GPU and run a two-sample Kolmogorov-Smirnov test on the cumulative density function (cdf) of the CPU and GPU squared errors. The results of the tests bring us to reject, at about the 30% level, the null hypothesis that CPU and GPU squared errors come from the same distribution, in favour of the alternative that CPU squared errors cdf is smaller than the GPU squared error cdf. Thus, we conclude that CPU and GPU give equivalent results under a statistical point of view up to a 30% significance level. One could expect that differences between GPU and CPU become more relevant for more difficult integration problems and operations involving division and matrix manipulations.

4.2. **Sequential Monte Carlo.** We also provide a raw estimate of the main differences in terms of precision for Sampling Importance Resampling (SIR), which is a standard Sequential Monte Carlo algorithm, applied to filtering

FIGURE 2. Histogram of the CPU-GPU RMSE differences for the SMC estimates of the SV.



of the hidden states of a nonlinear state space model with known parameters. We consider a stochastic volatility model

$$y_t = \exp\left\{\frac{1}{2}x_t\right\}\varepsilon_t,\ \varepsilon_t \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1),\ t=1,...,T \tag{11}$$

$$x_{t+1} = -0.01 + 0.9x_t + \eta_{t+1},\ \eta_{t+1} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,0.3) \tag{12}$$

where $\mathcal{N}(\mu,\sigma)$ denotes a normal distribution with mean μ and standard deviation σ. The parameter values are set for a typical weekly financial time series application (see Casarin and Marin, 2009). In Algorithm 1 we give the pseudo-code representation of the prediction and filtering steps of a SMC algorithm for SV model.

```
Algorithm 1. - SIR Particle Filter -
• At time $t = t_0$, for $i = 1,\dots,N$, simulate $x^i_{t_0} \sim p(x_{t_0})$ and set $\omega^i_{t_0} = 1/N$
• At time $t_0 < t \leq T-1$, given $\Xi_t = \{x^i_t, \omega^i_t\}_{i=1}^N$, for $i = 1,\dots,N$:
    (1) Simulate $\tilde{x}^i_{t+1} \sim \mathcal{N}(-0.01 + 0.9x^i_t, 0.3)$.
    (2) Update $\gamma^i_{t+1} \propto \omega^t_i \exp\left\{-\frac{1}{2}\tilde{x}^i_{t+1}\right\}\exp\left\{-\frac{1}{2}y^2_{t+1}\exp\{-\tilde{x}^i_{t+1}\}\right\}$.
    (3) Normalize $\tilde{\gamma}^i_{t+1} = \gamma^i_{t+1}/\sum_{j=1}^N \gamma^i_{t+1}$, $i = 1,\dots,N$
    (4) If $ESS_t < \kappa$ sample $x^i_{t+1}$, $i = 1,\dots,N$ from $\{\tilde{x}^i_{t+1}, \tilde{\gamma}^i_{t+1}\}_{i=1}^N$ and
        set $\omega^i_{t+1} = 1/N$, otherwise set $x^i_{t+1} = \tilde{x}^i_{t+1}$ and $\omega^i_{t+1} = \gamma^i_{t+1}$.
    (5) Set $\Xi_{t+1} = \{x^i_{t+1}, \omega_{t+1}\}_{i=1}^N$.
```

TABLE 2. GPU computing time (in seconds) for the SMC filter applied to a SV model, for different ESS threshold κ (columns). In the second row: percentage difference respect to the case $\kappa = 0.9999$.

	κ				
	0.9999	0.99995	0.99999	0.999995	0.999999
Time	6.906	6.925	6.935	7.008	7.133
Percentage	-	0.263	0.393	1.462	3.276

We fix $T = 100$, $N = 1000$ and $\kappa = 0.7$ and repeat $G = 1000$ times the Sequential Monte Carlo exercise. We compute Root Mean Square Errors (RMSE) between the true values x_t, $t = 1, ..., T$ and the simulated ones using a CPU code or a GPU one. Fig. 2 displays differences between the CPU and GPU RMSEs, where again the more concentrated the density is on the negative part of the support, the higher is the precision of the CPU with respect to the GPU. Concentration on the positive part corresponds to a GPU overperformance. The skewness of the histogram in 2 is 0.729, therefore there is a mild evidence in favour of GPU.

The resampling step of the SMC algorithm is performed on CPU and this may induce a loss of computational efficiency in the parallel GPU implementation of our DeCo package. We provides evidence of this fact for the simple SV model with known parameter values. Table 2 shows the GPU computing time for different values of the threshold, i.e. $\kappa = 0.9999$, 0.99995, 0.99999, 0.999995, 0.999999. The computational cost increase with the values of κ and the resampling frequency and the loss of information in the particle set increase as well. The potential drawback of a too low resampling frequency is the degeneracy of the particle set. In the SMC literature a value of κ about 0.7 is generally considered an appropriate one.

5. DIFFERENCES BETWEEN CPU AND GPU RESULTS II

Following BCRVD (2013) we compare the cases of Unbiased and Biased predictors and of complete and incomplete model sets using the DeCo code. We assume the true model is $\mathcal{M}_1 : y_{1t} = 0.1 + 0.6 y_{1t-1} + \varepsilon_{1t}$ with $\varepsilon_{1t} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$ and consider four experiments. We apply the DeCo package and use the GUI interface described in Appendix B to provide the inputs to the combination procedure.

Complete model set experiments. We assume the true model belongs to the set of models in the combination. In the first experiments the model set also includes two biased predictors: $\mathcal{M}_2 : y_{2t} = 0.3 + 0.2 y_{2t-2} + \varepsilon_{2t}$ and $\mathcal{M}_3 : y_{3t} = 0.5 + 0.1 y_{3t-1} + \varepsilon_{3t}$, with $\varepsilon_{it} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$, $i = 2, 3$. In the second experiment the complete model set includes also two unbiased predictors: $\mathcal{M}_2 : y_{2t} = 0.125 + 0.5y_{2t-2} + \varepsilon_{2t}$ and $\mathcal{M}_3 : y_{3t} = 0.2 + 0.2y_{3t-1} + \varepsilon_{3t}$, with $\varepsilon_{it} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$, $i = 2, 3$.

Incomplete model set experiments. We assume the true model is not in the model set. In the third experiment the model set includes two biased predictors: $\mathcal{M}_2 : y_{2t} = 0.3+0.2y_{2t-2}+\varepsilon_{2t}$ and $\mathcal{M}_3 : y_{3t} = 0.5+0.1y_{3t-1}+\varepsilon_{3t}$, with $\varepsilon_{it} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$, $i = 2, 3$. In the fourth experiment the model set includes unbiased predictors: $\mathcal{M}_2 : y_{2t} = 0.125 + 0.5y_{2t-2} + \varepsilon_{2t}$, $\mathcal{M}_3 : y_{3t} = 0.2 + 0.2y_{3t-1} + \varepsilon_{3t}$, with $\varepsilon_{it} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$, $i = 2, 3$.

We develop the comparison exercises with both 1000 and 5000 particles. Tables 3 report the time comparison (in seconds) to produce forecast combination for different experiments and different implementations. Parallel implementation on GPU NVIDIA Quadro K2000M is the most efficient, in terms of computing time, for all experiments. The gains are often substantial in terms of seconds (see Tab. 3, panel (a)), up to several hours when using 5000 particles (see Tab. 3, panel (b)). More specifically, the computational gain of the GPU implementation over parallel CPU implementation varies from 3 to 4 times for the Intel Core i7 and from 5 to 7 times for the Intel Xeon X3430. The overperformance of the parallel GPU implementation on sequential CPU implementation varies from 15 to 20 times when considering an Intel Xeon X3430 machine as a benchmark.

Figure 3 compares the weights for exercises 1 and 2. The weights follow a similar pattern, but there are discrepancies between them for some observations. Differences are larger for the median value than for the smaller and larger quantiles. The differences are, however, smaller and almost vanishes when one focuses on the predictive densities in Figure 4, which is the most important output of the density combination algorithm. We interpret the results as evidence of no economic and statistic significance of the differences between CPU and GPU draws.

Results are similar when focusing on the incomplete model set in Figures 5-6. Evidence does not change when we use 5000 particles.

Learning mechanism experiments. BCRVD (2013) document that a learning mechanism in the weights is crucial to identify the true model (in the case of complete model set) or the best model (in the case of incomplete model set) when the predictors are unbiased, see also left panels in Fig. 3-5. We repeat the two unbiased predictor exercises and introduce learning in the combination weights as discussed in Section 2. We set the learning parameters $\lambda = 0.95$ and $\tau = 9$. Table 4 reports the time comparison (in seconds) when using 1000 and 5000 particles filtered model probability weights. The computation time for DeCo increases when learning mechanisms are applied, in particular for the CPU. The GPU is from 10 to 50% slower than without learning, but CPU is from 2.5 to almost 4 times slower than previously. The GPU/CPU ratio, therefore, increases in favor of GPU

FIGURE 3. GPU and CPU 1000 particles filtered model probability weights for the complete model set. Model weights and 95% credibility region for models 1,2 and 3 (different rows).

Biased Predictors **Unbiased predictors**

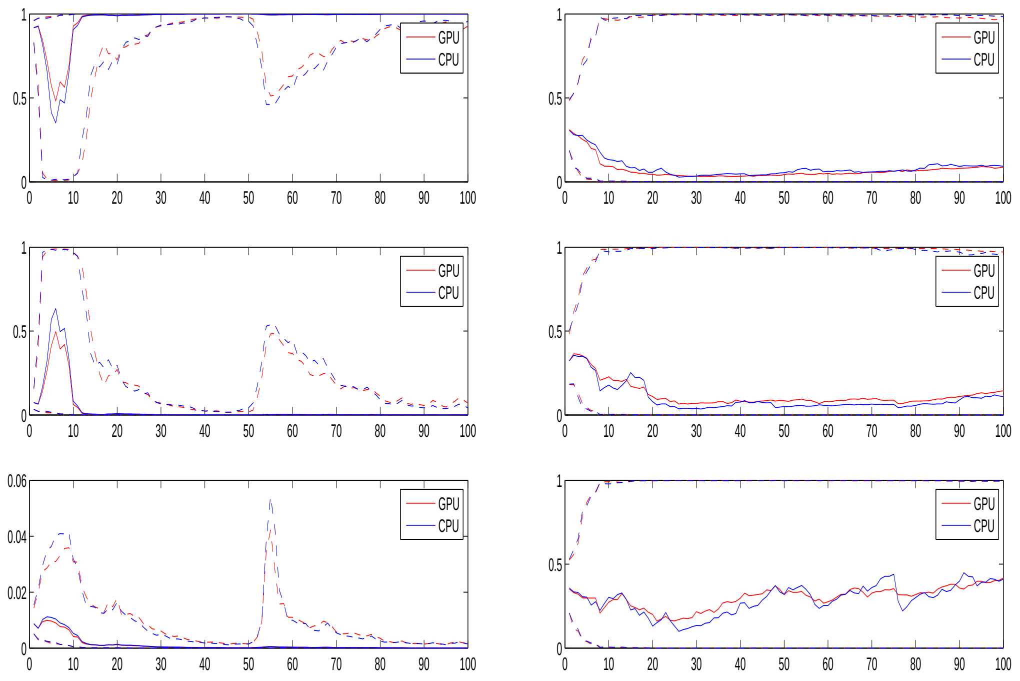


FIGURE 4. GPU and CPU 1000 particles filtered density forecasts for the complete model set. Mean and 95% credibility region of the combined predictive density.

Biased Predictors **Unbiased predictors**

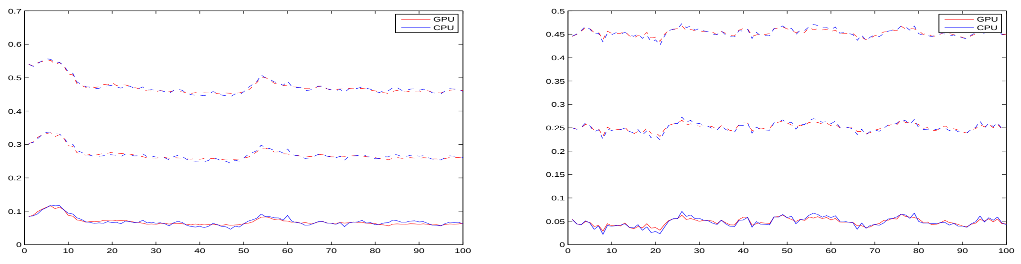


FIGURE 5. GPU and CPU 1000 particles filtered model probability weights for the incomplete model set. Model weights and 95% credibility region for models 1,2 and 3 (different rows).

Biased Predictors **Unbiased predictors**

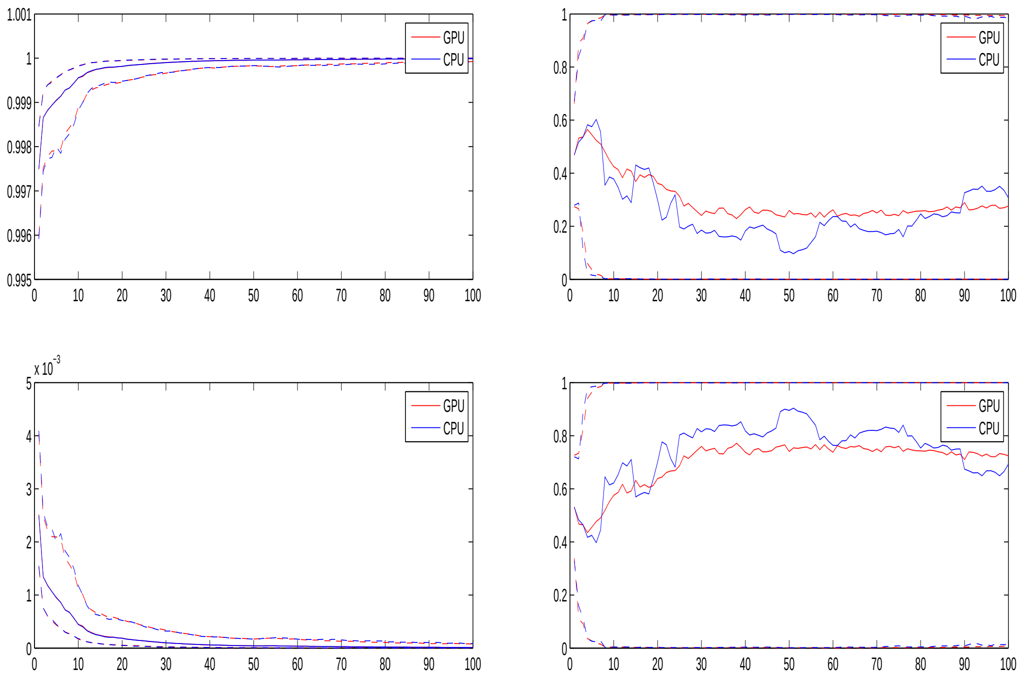


FIGURE 6. GPU and CPU 1000 particles filtered density forecasts for the incomplete model set. Mean and 95% credibility region of the combined predictive densities.

Biased Predictors **Unbiased predictors**

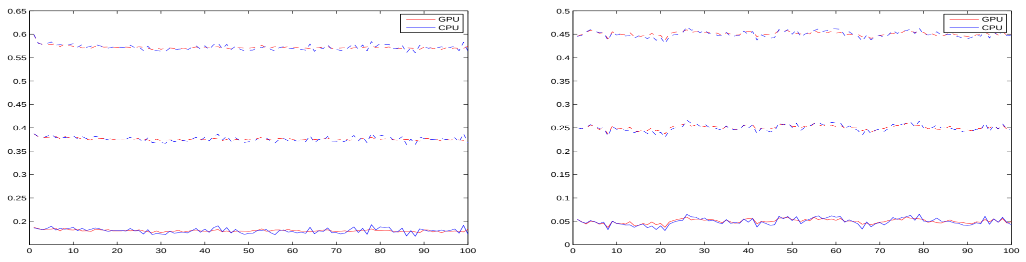


TABLE 3. Density combination computing time in seconds. Rows: different simulation exercises. Columns: parallel GPU (*p-GPU*) and parallel CPU (*p-CPU-i7*) implementations on GPU NVIDIA Quadro K2000M with CPU Intel Core i7-3820QM, 3.7GHz; parallel CPU (*p-CPU-Xeon*) and sequential CPU (*CPU-Xeon*) implementations on Intel Xeon X3430 4core, 2.40GHz. In parenthesis: efficiency gain in terms of CPU/GPU times ratio.

(a) 1000 Particles

	p-GPU	p-CPU-i7	p-CPU-Xeon	CPU-Xeon
Complete Model Set				
1 Biased Predictors	699	2780	5119	11749
		(3.97)	(7.32)	(16.80)
2 Unbiased Predictors	660	2047	5113	11767
		(3.10)	(7.75)	(17.83)
Incomplete Model Set				
3 Biased Predictors	671	2801	5112	11635
		(4.17)	(7.62)	(17.34)
4 Unbiased Predictors	687	2035	5098	11636
		(2.96)	(7.42)	(16.94)

(b) 5000 particles

	p-GPU	p-CPU-i7	p-CPU-Xeon	CPU-Xeon
Complete Model Set				
1 Biased Predictors	4815	15154	26833	64223
		(3.15)	(5.57)	(13.34)
2 Unbiased Predictors	5302	15154	26680	63602
		(2.86)	(5.03)	(12.00)
Incomplete Model Set				
3 Biased Predictors	4339	13338	26778	64322
		(3.07)	(6.17)	(14.82)
4 Unbiased Predictors	4581	13203	26762	63602
		(2.88)	(5.84)	(13.88)

with GPU computation from 5 to 70 times faster depending on the alternative CPU machine considered. The DeCo codes with learning have some if commands related to the minimum numbers of observations necessary to initiate the learning which increases computational time substantially. The parallelization in GPU is more efficient and these if commands play a minor role. We expect that the gain might increase to several hundred of times when using parallelization on cluster of computers.

TABLE 4. Density combination computing time in seconds. Rows: different simulation exercises. Columns: parallel GPU (*p-GPU*) and parallel CPU (*p-CPU-i7*) implementations on GPU NVIDIA Quadro K2000M with CPU Intel Core i7-3820QM, 3.7GHz; parallel CPU (*p-CPU-Xeon*) and sequential CPU (*CPU-Xeon*) implementations on Intel Xeon X3430 4core, 2.40GHz. In parenthesis: efficiency gain in terms of CPU/GPU times ratio.

	p-GPU	p-CPU-i7	p-CPU-Xeon	CPU-Xeon
(a) 1000 Particles				
2 Complete Model Set	755	7036	14779	52647
		(9.32)	(19.57)	(69.73)
4 Incomplete Model Set	719	6992	14741	52575
		(9.72)	(20.49)	(73.08)
(b) 5000 particles				
3 Complete Model Set	7403	35472	73402	274220
		(4.79)	(9.92)	(37.04)
4 Incomplete Model Set	7260	35292	73256	274301
		(4.86)	(10.09)	(37.78)

6. EMPIRICAL APPLICATION

As a further check of the performance of the DeCo code, we compare the CPU and GPU versions in the macroeconomic application developed in BCRVD (2013). We consider $K = 6$ time series models to predict US GDP growth and PCE inflation: an univariate autoregressive model of order one (AR); a bivariate vector autoregressive model for GDP and PCE, of order one (VAR); a two-state Markov-switching autoregressive model of order one (ARMS); a two-state Markov-switching vector autoregressive model of order one for GDP and inflation (VARMS); a time-varying autoregressive model with stochastic volatility (TVPARSV); and a time-varying vector autoregressive model with stochastic volatility (TVPVARSV). Therefore, the model set includes constant parameter univariate and multivariate specification; univariate and multivariate models with discrete breaks (Markov-Switching specifications); and univariate and multivariate models with continuous breaks. These are typical models applied in macroeconomic forecasting; see, for example, Clark and Ravazzolo (2012), Korobilis (2011) and D'Agostino et al. (2011).

TABLE 5. Computing time and forecast accuracy for the macro-economic application for the GPU (column GPU) and CPU (column CPU) implementations. Rows: Time: time in seconds to run the exercise in seconds; RMSPE: Root Mean Square Prediction Error; CW: p-value of the Clark and West (2007) test; LS: average Logarithmic Score over the evaluation period; CRPS: cumulative rank probability score; LS p-value and CRPS p-value: Harvey et al. (1997) type of test for LS and CRPS differentials respectively.

	GDP		Inflation	
	GPU	CPU	GPU	CPU
Time	1249	6923	-	-
RMSPE	0.634	0.637	0.255	0.256
CW	0.000	0.000	0.000	0.000
LS	-1.126	-1.130	0.251	0.257
p-value	0.006	0.005	0.021	0.022
CRPS	0.312	0.313	0.112	0.112
p-value	0.000	0.000	0.000	0.000

We evaluate the two combination methods by applying the following evaluation metrics: Root Mean Square Prediction Errors (RMSPE), Kullback Leibler Information Criterion (KLIC) based measure, the expected difference in the Logarithmic Scores (LS) and the Continuous Rank Probability Score (CRPS). Accuracy statistics and related tests (see BCRVD (2013)) are used to compare the forecast accuracy.

Table 5 reports results for the multivariate combination approach. For the sake of brevity, we just present results using parallel GPU and the best parallel CPU Intel Core i7-3820QM machine. We also do not consider learning mechanism in the weights. GPU is substantially faster, almost 5.5 times faster than CPU, reducing the computational time of more than 5000 seconds. GPU is therefore performing relatively better in this exercise than in the previous simulation exercises (without learning mechanisms). The explanation relies on the larger set of models and the multivariate application. The number of simulation has increased substantially and CPU starts to hit physical limits, slowing down the computation and extending time. GPU has not binding limits and just double the time of simulation exercises with a univariate series and the same number of draws and particles.[3] This suggests that GPU might be an efficient methodology to investigate when averaging large set of models.

[3]Unreported results show that GPU is more than 36 times faster than sequential CPU implementation on Intel Xeon X3430 4core.

Accuracy results for CPU and GPU combinations are very similar and just differ after the third decimals, confirming previous intuitions that the two methods are not necessarily numerical identical, but provide identical economic and statistical conclusions.[4] The combination approach is statistically superior to the AR benchmark for all the three accuracy measures we implement.

7. Conclusion

This paper introduces the Matlab package DeCo (Density Combination) based on parallel Sequential Monte Carlo simulations to combine density forecasts with time-varying weights and different choices of scoring rule.

The package is easy to use for a standard Matlab user and to facilitate promulgation we have implemented a GUI user interface, which just requires a few input parameters. The package takes full advantage of recent computer hardware progresses and uses banks of parallel SMC algorithms for the density combination both using multi-core CPU and GPU implementation.

The DeCo GPU version is faster than the CPU version up to 70 times and even more for larger set of models. More specifically, our simulation and empirical exercises were conducted using a commercial notebook with CPU Intel Core i7-3820QM and GPU NVIDIA Quadro K2000M, and Matlab 2012b version, and show that DeCo GPU version is faster than the parallel CPU version, up to 5.5 times, when using a i7 CPU machine and the Parallel Computing Toolbox. In the comparison between GPU and non-parallel CPU implementations, the differences between GPU and CPU time, increase up to almost 70 times, when using a standard CPU processor, such as quad-core Xeon. At this stage of our research, we do not have extreme versions of the graphical cards, such as GTX cards, to provide further evidence. All comparisons have been implemented using double precision for both CPU and GPU versions. If an application allows for lower degree of precision, for example when the interest is on point forecasting, and the use of single precision Lee et al. (2010) and Geweke and Durham (2012) document massively gains (up to 500 times) to a single-threaded CPU code. Matlab just works on double precision.

We also document that the CPU and GPU versions do not necessarily provide the exact same numerical solutions to our problems, but differences are not economically and statistically significant. Therefore, users of DeCo might choose between the CPU and GPU versions depending on the available and preferred clusters.

Finally, Mathworks has recently bought Accelereyes' Jacket engine, a library to run Matlab applications in GPU. Mathworks has not yet a "for"

[4]Numbers for the CPU combination differ marginally from those in Table 5 in BCRVD (2013) due to the use of a different Matlab version, different generator numbers and parallel tooling functions. Also in this case, the differences are numerically, but very small and therefore have not any economic and statistical significance.

loop command for GPU computation in Matlab, but Accelereyes has it. We expect Mathworks to incorporate this command and DeCo GPU version will benefit enormously by running some for loops directly in GPU without to transfer data to the CPU.

References

Aldrich, E. M. (2013). Massively parallel computing in economics. Technical report, University of California, Santa Cruz.

Aldrich, E. M., Fernández-Villaverde, J., Gallant, A. R., and Rubio Ramırez, J. F. (2011). Tapping the Supercomputer Under Your Desk: Solving Dynamic Equilibrium Models with Graphics Processors. *Journal of Economic Dynamics and Control*, 35:386–393.

Bates, J. M. and Granger, C. W. J. (1969). Combination of Forecasts. *Operational Research Quarterly*, 20:451–468.

Billio, M., Casarin, R., Ravazzolo, F., and van Dijk, H. K. (2013). Time-varying Combinations of Predictive Densities Using Nonlinear Filtering. *Journal of Econometrics*, forthcoming.

Casarin, R. and Marin, J. M. (2009). Online data processing: Comparison of Bayesian regularized particle filters. *Electronic Journal of Statistics*, 3:239–258.

Chong, Y. and Hendry, D. F. (1986). Econometric Evaluation of Linear Macroeconomic Models. *The Review of Economic Studies*, 53:671 – 690.

Clark, T. and Ravazzolo, F. (2012). The macroeconomic forecasting performance of autoregressive models with alternative specifications of time-varying volatility. Technical report, FRB of Cleveland Working Paper 12-18.

Clark, T. and West, K. (2007). Approximately Normal Tests for Equal Predictive Accuracy in Nested Models. *Journal of Econometrics*, 138(1):291–311.

Creal, D. (2009). A survey of sequential Monte Carlo methods for economics and finance. *Econometric Reviews*, 31(3):245–296.

Creel, M. (2005). User-Friendly Parallel Computations with Econometric Examples. *Computational Economics*, Vol. 26:pp. 107 – 128.

Creel, M. and Goffe, W. L. (2008). Multi-core cpus, Clusters, and Grid Computing: A Tutorial. *Computational Economics*, Vol. 32:pp. 353 – 382.

D'Agostino, A., Gambetti, L., and Giannone, D. (2011). Macroeconomic forecasting and structural change. *Journal of Applied Econometrics*, forthcoming.

Doucet, A., Freitas, J. G., and Gordon, J. (2001). *Sequential Monte Carlo Methods in Practice*. Springer Verlag, New York.

Dziubinski, M. P. and Grassi, S. (2013). Heterogeneous Computing In Economics: A Simplified Approach. *Computational Economics*, forthcoming.

Geweke, J. and Amisano, G. (2010). Optimal prediction pools. *Journal of Econometrics*, 164(2):130–141.

Geweke, J. and Durham, G. (2012). Massively Parallel Sequential Monte Carlo for Bayesian Inference. Working papers, National Bureau of Economic Research, Inc.

Gneiting, T. and Raftery, A. E. (2007). Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102:359–378.

Granger, C. W. J. and Ramanathan, R. (1984). Improved Methods of Combining Forecasts. *Journal of Forecasting*, 3:197–204.

Gregory, K. and Miller, A. (2012). *Accelerated Massive Parallelism with Microsoft Visual C++*. Microsoft Press, USA.

Hall, S. G. and Mitchell, J. (2007). Combining density forecasts. *International Journal of Forecasting*, 23:1–13.

Harvey, D., Leybourne, S., and Newbold, P. (1997). Testing the equality of prediction mean squared errors. *International Journal of Forecasting*, 13:281–291.

IEEE (2008). *IEEE 754 - 2008. IEEE 754 - 2008 Standard for Floating-Point Arithmetic*. IEEE.

Jore, A. S., Mitchell, J., and Vahey, S. P. (2010). Combining forecast densities from VARs with uncertain instabilities. *Journal of Applied Econometrics*, 25(4):621–634.

Korobilis, D. (2011). VAR Forecasting Using Bayesian Variable Selection. *Journal of Applied Econometrics*, forthcoming.

Lee, A., Christopher, Y., Giles, M. B., Doucet, A., and Holmes, C. C. (2010). On the Utility of Graphic Cards to Perform Massively Parallel Simulation with Advanced Monte Carlo Methods. *Journal of Computational and Graphical Statistics*, 19:4:769–789.

Legland, F. and Oudjane, N. (2004). Stability and uniform approximation of nonlinear filters using the Hilbert metric and application to particle filters. *Annals of Applied Probability*, 14(1):144–187.

LeSage, J. P. (1998). Econometrics: Matlab toolbox of econometrics functions. Statistical Software Components, Boston College Department of Economics.

Liu, J. and Chen, R. (1998). Sequential Monte Carlo methods for dynamical system. *Journal of the American Statistical Association*, 93:1032–1044.

Liu, J. S. and West, M. (2001). Combined parameter and state estimation in simulation based filtering. In Doucet, A., de Freitas, N., and Gordon, N., editors, *Sequential Monte Carlo Methods in Practice*. Springer-Verlag.

Morozov, S. and Mathur, S. (2011). Massively Parallel Computation Using Graphics Processors with Application to Optimal Experimentation in Dynamic Control. *Computational Economics*, pages 1 – 32.

Musso, C., Oudjane, N., and Legland, F. (2001). Improving regularised particle filters. In Doucet, A., de Freitas, N., and Gordon, N., editors, *Sequential Monte Carlo Methods in Practice*. Springer-Verlag.

Robert, C. P. and Casella, G. (2004). *Monte Carlo Statistical Methods*. Springer, Berlin, Second Edition.

Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal, http://www.gotw.ca/publications/concurrencyddj.htm*.

Sutter, H. (2011). Welcome to the Jungle. *http://herbsutter.com/welcome-to-the-jungle/*.

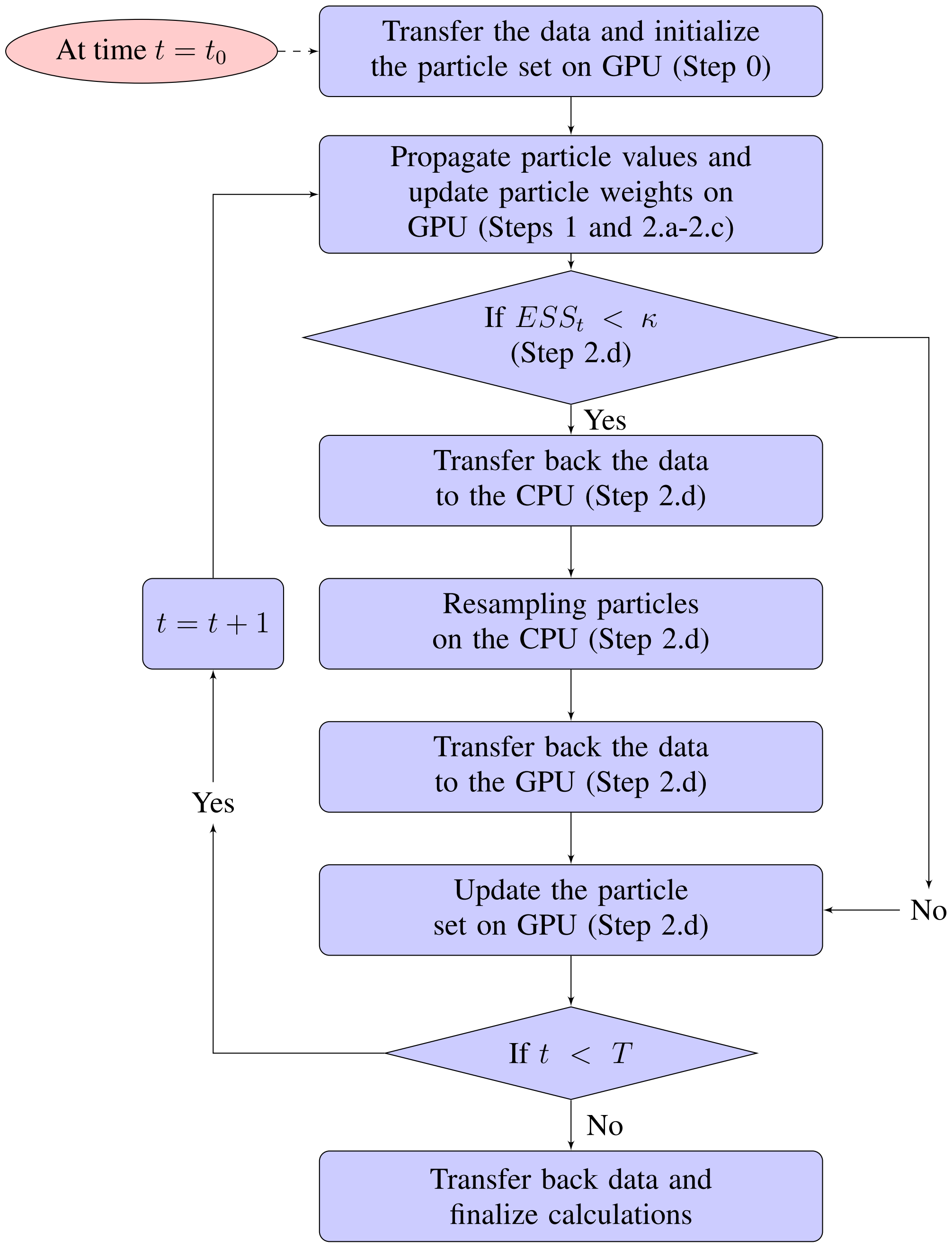
Swann, C. A. (2002). Maximum Likelihood Estimation Using Parallel Computing: An Introduction to MPI. *Computational Economics*, Vol. 19:pp. 145 – 178.

Terui, N. and van Dijk, H. K. (2002). Combined Forecasts from Linear and Nonlinear Time Series Models. *International Journal of Forecasting*, 18:421–438.

Whitehead, N. and Fit-Florea, A. (2011). Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. *NVIDIA Tech Report, http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf*.

APPENDIX A. FLOW-CHART OF GPU DECO PACKAGE

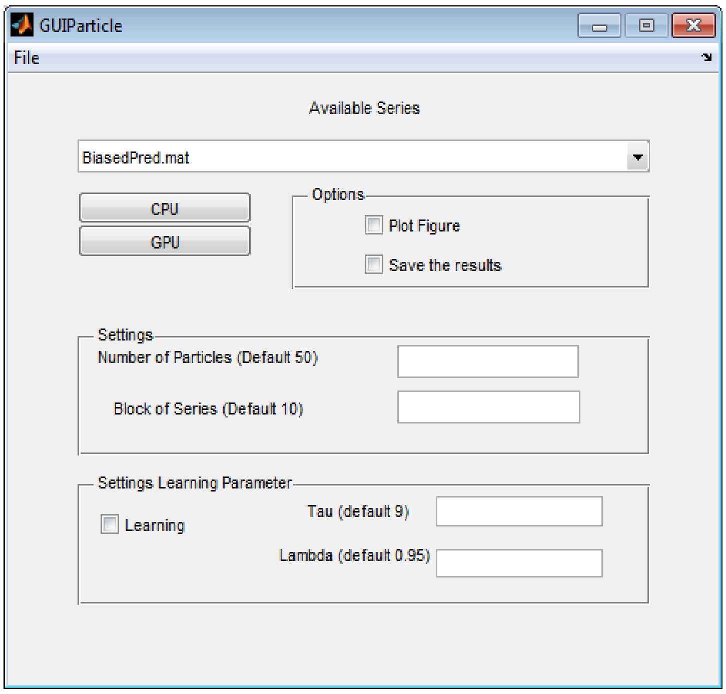
FIGURE 7. Flow chart of the parallel SMC filter given in Section 2



At time $t = t_0$
Transfer the data and initialize the particle set on GPU (Step 0)
Propagate particle values and update particle weights on GPU (Steps 1 and 2.a-2.c)
If $ESS_t < \kappa$ (Step 2.d)
Yes
Transfer back the data to the CPU (Step 2.d)
Resampling particles on the CPU (Step 2.d)
Transfer back the data to the GPU (Step 2.d)
Update the particle set on GPU (Step 2.d)
No
If $t < T$
Yes
$t = t + 1$
No
Transfer back data and finalize calculations

APPENDIX B. THE GUI USER INTERFACE

FIGURE 8. The graphical user interface of the DeCo package


GUIParticle

File

Available Series

BiasedPred.mat

CPU

GPU

Options

Plot Figure

Save the results

Settings

Number of Particles (Default 50)

Block of Series (Default 10)

Settings Learning Parameter

Learning

Tau (default 9)

Lambda (default 0.95)

The Figure 8 shows the GUI of the DeCo package, that contains all the necessary inputs for our program. The ListBox loads and displays the available dataset in the directory $Dataset$. The figure shows, as example, the dataset *BiasedPred.mat*. The number of particles and the block of selected series are chosen in the edit box “Settings”. The default values are set to 50 for the particles and 10 for the block of series, this is only valid for the GPU. The box *options* contains commands for plotting and saving results. The results are saved in the directory $OutputCPU$ or $OutputGPU$ depending on the type of calculation chosen. Finally the botton “CPU” starts the corresponding CPU program and the botton “GPU” executes the program on the GPU. The box "Setting Learning Parameter" allows the user to perform the calculation with or without learning, see section 5. When the option Learning is chosen, the edit box allows to set the learning parameter, the default values are $\lambda = 0.95$ and $\tau = 9$.

Some considerations are in order. First, the CPU is already implemented in parallel form. The user has to start a parallel session in Matlab by typing the command $matlabpool$ *open* in the Matlab main window. Please refer to Matlab online help. Second, the dataset accepted by the program is *mat* format and has the following form. It includes two variables, the first one is defined as vY and it contains a $(T \times L)$ matrix of the the variables $\{\mathbf{y}_t\}_{t=1}^{T}$ to be forecasted, where T is the number of 1-step ahead forecasts and L the size of observable variables to forecast. The second one is a $4-D$ matrix defined mX with the following dimensions (T, M, L, KL) , where M is the size of i.i.d. samples from the predictive densities, and KL the number of 1-step ahead predictive densities. Finally, the user might apply different learning mechanisms based on other scoring function that the one applied and discussed in Section 2 should change the function “PFCore.m”.