Efficient Parallelization of Macroeconomic Models: A Matlab Executable Approach

Liang Shi*

July 2024

Abstract

Grid-based numerical solutions of macroeconomic models can be time-consuming to obtain, posing a bottleneck in macroeconomic research, such as in the case of the sovereign default model à la Arellano (2008). Using the widely compatible value function iteration algorithm, I leverage Matlab Executable (Mex) to generate and compile parallelized C (Mex parfor) and CUDA code (Mex CUDA), thereby efficiently deriving numerical solutions for a sovereign default model. Mex parfor provides over a 20-fold speedup compared to serial Matlab code, achieving the performance similar to GFortran with OpenMP. Meanwhile, Mex CUDA can offer a 50-fold acceleration in double precision and an impressive 600-fold acceleration in single precision, again comparable to the performance of OpenACC with NVIDIA HPC SDK. Unlike Mex parfor, the advantage of Mex CUDA tends to grow with the size of workload. Appropriate algorithms and pragmas are important to create kernels making the most of GPU.

Keywords: Parallelized Computation, Matlab Mex, Fortran, OpenMP, OpenACC, Sovereign Default, Macroeconomics

JEL Codes: C63, F34, F41

*Department of Economics, University of Essex, CO4 3SQ, Colchester, United Kingdoms. Email: shiliang369econ@gmail.com. I thank Christoph Görtz for insightful comments. Colour should be used for all figures in print. Declarations of interest: none. All errors are my own.

1 Introduction

Computational resources are often a limiting factor in macroeconomic research that is based on complex dynamic stochastic general equilibrium models, which feature non-linearity and render efficient perturbation methods unfeasible. A common choice for solving these models is grid-based recursive algorithms, such as the widely compatible Value Function Iteration (VFI) methods. Unfortunately, these methods tend to be slow, especially when the number of grids is large. As modern computer chips' single-core performance approaches physical limits, utilizing multiple computing cores becomes increasingly important for enhancing computational performance. Following this trend, economists have been accelerating model solving and simulation by employing parallelization (multithreading) techniques. In this paper, using the sovereign default model (Arellano, 2008) as an example, I discuss parallelization with Matlab Executable (Mex).

Specifically, this Mex method generates C/CUDA code from functions written in native Matlab language, and users interacts with the code generation process within the Matlab's integrated development environment (IDE). The C/CUDA code generation is guided by directives/pragmas in the Matlab function files, allowing users to bypass the learning curve of complex C/CUDA languages, avoid time-consuming coding and debugging, and maintain convenience in data analysis. The generated code is then compiled with high-level optimization, utilizing third-party compilers to fully leverage hardware potential. The generated Mex function can be called within Matlab scripts as external functions, with inputs and outputs managed within the Matlab environment. Since a large community in economics intensively uses Matlab for model solving and simulation, this Mex method effectively combines Matlab's user-friendliness with the efficiency of compiled C/CUDA languages to accelerate program execution.

I demonstrate how to generate parallelized Mex code from Matlab code. Both CPU-based (Mex parfor) and GPU-based (Mex CUDA) parallelizations are discussed. I find the speed of Mex parfor is over 20 times faster than the non-compiled sequential execution of the same program, referred to as Matlab serial. Mex CUDA provides an even more desirable speedup. With data in double-precision, it can outperform Matlab serial by over 50 times, and this advantage expands with increasing workload. This advantage grows to 660 times when I loosen the data precision to be single-precision, without loss of accuracy in solutions. I show that Mex-based parallelization is as efficient as the renowned GNU Fortran (GFortran) with Open Multi-Processing (OpenMP) when tasks run on a CPU, and as efficient as the NVIDIA HPC SDK Fortran (NvFortran)

with Open Accelerators (OpenACC) on a GPU. These languages, libraries, and compilers are designed for scientific computation: They compile highly optimized machine code and efficiently utilize memory and computing cores. Therefore, the comparison confirms the Mex method is a convenient and competitive approach to applying multithreading in economics research.

While the benchmark performance of Mex parfor and Mex CUDA is evaluated using a sovereign default model, this paper also provides results for a basic stochastic growth model, which has been discussed in economic computation studies such as Aldrich et al. (2011) and Aruoba and Fernández-Villaverde (2015). This alternative model is simpler than the benchmark sovereign default model in terms of code complexity, computational workload, and memory allocation. Compared to the benchmark model, Mex parfor improves its speedup, while Mex CUDA provides relatively smaller speed gains. Both methods perform comparably with their respective Fortran counterparts.

This paper is situated within the field of computational economics, specifically addressing the use of multithreading with CPUs and GPUs. Using VFI and binary search algorithm, Aldrich et al. (2011) report a 300-fold performance improvement of CUDA over a C++ serial CPU implementation when solving a stochastic growth model. The choice of algorithm is importance for performance. With an alternative grid search and Howard improvement method, the outperformance of CUDA shrinks to less than three-fold. Morozov and Mathur (2012) analyze CUDA's performance in solving imperfect information dynamic models, finding approximately a 15-fold improvement of GPU over single-threaded CPU implementations using Fortran. Similar evaluations of multithreading are discussed in Aldrich (2014) and Fernández-Villaverde and Valencia (2018). Generally, implementations using compiled languages such as C, C++, Fortran, and CUDA are notably more efficient in terms of runtime and speedup compared to single-threaded implementations. In contrast, interpreted languages such as Matlab, Julia, Python, and R exhibit significantly lower performance.

However, interpreted programming languages have been enjoying significant popularity. One reason is that languages such as Matlab are less verbose, more flexible, and easier to learn compared to compiled languages, which has led to their widespread adoption among economists. Additionally, the overall time investment is a crucial factor. While the execution time of programs written in interpreted languages is typically longer, the time required for writing, debugging, and performing subsequent data analysis is often considerably shorter. For instance, Matlab offers a

convenient code editing and execution environment, plotting capabilities, and a large collection of pre-written routines (Coleman et al., 2021). Consequently, hybrid approaches that combine the ease of use of interpreted languages with the performance benefits of compiled languages, such as Mex, are becoming increasingly appealing (Aruoba and Fernández-Villaverde, 2015).

However, the use of multithreading within hybrid environments like Mex has been underdiscussed in the economics community. Maliar and Maliar (2014) demonstrate a significant speedup achievable with Matlab's 'parfor' for native multithreading on CPU cores, but comparable Mex and native Matlab GPU implementations do not work well. Richter et al. (2014) report an eight-fold speedup in a real business cycle model and a 24-fold speedup in a New Keynesian model by employing Mex multithreading on the CPU. In their approach, functions are written in Fortran, with Mex used to integrate them into Matlab's IDE. In contrast, this paper presents a method for generating executables in C and CUDA based on native Matlab code and a few pragmas, thus avoiding the need to master an additional language. To the best of my knowledge, this is the first study to demonstrate the effective use of Mex-based multithreading, including both CPU and GPU implementations, within the field of economics.

This paper is also related to the sovereign default models (Arellano, 2008) and the associated computational discussions. Hatchondo et al. (2010) emphasize the importance of robust numerical implementations and the accuracy of the resulting solutions. Arellano et al. (2016) introduce the envelope condition method, which offers significant speed gains over conventional VFI but is prone to convergence issues. Deng et al. (2022) explore GPU-based solutions using Julia CUDA, achieving a tenfold speedup over serial CPU methods implemented in C++. The sovereign default model is selected as a benchmark due to its non-linearity, which substantially slows down numerical solving and thus makes the application of multithreading particularly advantageous. The multithreading methods discussed in this paper are applicable to other macroeconomic models requiring grid-based methods, such as those involving heterogeneous agents, sudden stops, search and matching, and zero-lower bounds.

The rest of this paper is organized as follows. Section 2 presents the model, the algorithms of VFI, and the setup of Mex-based multithreading. Section 3 compares the Mex results with Fortran counterparts, and discusses the differences in Mex parfor and Mex CUDA. Section 4 a sensitivity analysis where I apply the same methods to an alternative stochastic growth model. Section 5 draws the conclusions.

2 Model, Algorithm, and Coding

In this section discuss the sovereign default model with one-period debt, which is employed for evaluating computational performance. After detailing the model, I explain the efficient algorithm to solve the model, and then the implementation of Mex-based CPU and GPU multithreading to enhance computational speed.

2.1 The Sovereign Default Model

The model follows the quantitative framework established by Arellano (2008). In each period, the economy receives an exogenous endowment y, the realization of which follows an AR(1) process:

$$\log(y) = \rho_y \log(y) + \varepsilon^y, \qquad \varepsilon^y \sim i.i.N(0, \sigma^y) \tag{1}$$

A social planner determines whether to default ($\mathcal{D} = 1$) or not ($\mathcal{D} = 0$) by evaluating

$$V(y, b) = \max_{d=0,1} \Big\{ (1 - d(y, b))V^p(y, b) + d(y, b)V^d(y) \Big\} \tag{2}$$

V^p is the value of repaying debts. The planner optimizes by choosing the amount of debt issuance b':

$$V^p(y, b) = \max_{b' \geq 0,\, c \geq 0} \big\{ u(c) + \beta \mathbb{E}_{y'|y} V(y', b') \big\} \tag{3}$$

subject to a budget constraint $c = y - b + q(y, b')b'$, where q is the issuance price. V^d is the value of default, which results in exclusion from international borrowing:

$$V^d(y) = u(y - L(y)) + \beta \mathbb{E}_{y'|y} \Big[(1 - \mu)V^d(y) + \mu V(y', 0) \Big] \tag{4}$$

where $L(y)$ is an endowment loss due to default. After default, there is a probability μ of debt forgiveness, allowing re-entry into sovereign debt issuance. Foreign lenders incorporate the expected debt issuance and default probability into debt pricing:

$$q(y, b') = \mathbb{E}_{y'|y} \Big[1 - d(y', b') \Big] / (1 + r^*) \tag{5}$$

where r^* denotes the world risk-free interest rate as a discount rate.

Definition 1 (Recursive equilibrium)**.** Given y and b, the recursive equilibrium of the sovereign

default model is characterized by a set of value functions $V^p(y,b)$, $V^d(y)$ and $V(y,b)$, the optimal debt issuance $\mathcal{B} \equiv b'(y,b)$, default decision $d(y,b)$, and a debt price $q(y,b')$. Given price $q(y,b')$ and budget constraint, we solve the social planner's problem, obtain value functions, and the default choice; Given $d(y,b)$ the $q(y,b')$ satisfies the rule (5).

Grid-based VFI is a common approach for solving sovereign default models. Specifically, the spaces of endowment and debt states are discretized into grid points $\mathbb{Y} = \{y_i\}_{i=1}^{N_y}$ and $\mathbb{B} = \{b_i\}_{i=1}^{N_b}$. The transition probability of the exogenous endowment $\mathbb{E}_{y'|y}$ is approximated by an (N_y, N_y) matrix. The grid for debt policy is the same as that for the debt state. The values of repayment and default, debt prices, and all relevant variables are established on the coordinates of the endowment, debt, and debt policy grids. In this paper, we do not employ off-the-grid methods to locate the optimal debt policy. Readers interested in off-the-grid methods can refer to Hatchondo et al. (2010), where the robustness of Chebyshev collocation and spline interpolation methods is discussed. However, it is worth mentioning that these methods are not universally applicable. For instance, they may encounter convergence challenges when long-maturity debts are incorporated (Chatterjee and Eyigungor, 2012).

The nature of the sovereign default model precludes the use of fast perturbation methods, as demonstrated in studies such as Sims (2002) and Schmitt-Grohé and Uribe (2004). Several features lead to this restricted choice of algorithm. First, the default decision depends on the comparison between heterogeneous value functions representing repayment and default, resulting in kinks where derivatives are not feasible. Second, the debt price is defined by the expected choices of default and debt issuance, which are endogenous and depend on the grids of y and b. The policy of debt $\mathcal{B}$ is also difficult to linearly approximate. Third, defaults typically occur at the lower end of the endowment distribution and the higher end of debt outstanding, characterizing default risk as a tail risk. These characteristics necessitate the use of grid-based global methods, with VFI being one of them.

2.2 The Algorithm to implement VFI

In this section, I discuss the algorithm that implements the VFI method. The algorithm includes an outer "while" loop that controls the iteration of value and price functions and determines the convergence of the iteration, as illustrated in Algorithm 1. This single while-loop method, as employed by Hatchondo et al. (2010), is much more efficient than the multiple while-loop

Algorithm 1: The VFI while loop for solving the sovereign default model

Data: Initial guesses $V_0^d(y)$, $V_0^p(y,b)$, $V_0(y,b)$, and $q_0(y,b')$; while-loop parameters tol and Δ.

Result: V^d, V^p, V, q and $\mathcal{B}$.

while $\Delta > tol$ **do**

1. Given y, b, q_0, and possible b', calculate the consumption: $c = b' \cdot q - b + y$
2. Calculate utility $u(c)$, excluding $c < 0$;
3. Find $V^p \coloneqq \max_{b'}\{u(c) + \beta \mathbb{E} V_0^p\}$;
4. Calculate $V^d \coloneqq u(y - L(y)) + \beta \mathbb{E}(\mu V_0(y_i, 0) + (1-\mu)V_0^p)$;
5. $d(y_i, b_i) \coloneqq 1$ where $V^p(y_i, b_i) < V^d(y_i)$, otherwise $d(y_i, b_i) \coloneqq 0$;
6. Derive the price $q \coloneqq (1 - \mathbb{E}d)/(1 + r^*)$;
7. Update the value of default choice $V \coloneqq \max\{V^p, V^d\}$;
8. $\Delta \coloneqq \| V^p - V_0^p \| + \| V^d - V_0^d \| + \| q - q_0 \|$;
9. Update value functions and price $V_0^p = V^p$, $V_0^d = V^d$, $q_0 = q$

end

Repeat step 1, 2, 3 to find the optimal debt issuance $\mathcal{B}$

methods used in earlier VFI implementations of sovereign default models, such as Aguiar and Gopinath (2006). The majority of the computational workload occurs in steps 1 to 3. For each coordinate of the state grids (y_i, b_i), if the country repays, the utility must be evaluated for each possible value of debt issuance $b'i$. In other words, all operations in these three steps are performed on $N_y \times N_b \times N_b$ elements. Meanwhile, steps 5 to 9 are performed on a much smaller set of $N_y \times N_b$ elements, and step 4 only on N_y elements because V^d corresponds to zero debt outstanding. Therefore, efficient parallelization must address the bottleneck in steps 1 to 3[1].

```
1  evp = betta*pdfy*vp;  % pdfy is the prob transition matrix of y
2  for iy = 1:ny         % grids for y
3    for ib = 1:nb       % grids for b
4        tmpmax = - Inf ;
5      for i = 1:nb      % grids for debt issuance
6        c1 =  b(i)*q(iy,i) - b(ib) + y(iy) ;  % budget constraint
7        if c1 <= 0; c1 = - Inf ;              % exclude c<=0
8          else  c1 = (c1^(1-sigg)-1)/(1-sigg) + evp(iy,i) ; end
9        if tmpmax < c1; tmpmax = c1 ; end     % find the max
10     end
11     vp1(iy,ib) = tmpmax ;    % save the maximum values
12   end
13 end
```

Listing 1: Matlab codes for nested loops (corresponding to steps 1-3 in algorithm 1) calculating the V^p. Othe parts of the codes (for example the calculation of q and V^d) are reported in the Appendix.

I present the Matlab codes for steps 1-3 of the Algorithm 1. The implementation consists of three layers of for-loops to explicitly visit all the grid points for y, b, and b'. The calculation of $\beta \mathbb{E} V_0^p$ ('evp' in the first line) is out of these loops because it is irrelevant to b' and can exploit the efficiency in matrix multiplication. The "if" statements in line 5 and line 13 find the maximum welfare by searching all policy grids[2]. The inclusion of these loops provides remarkable efficiency for Matlab serial execution, which may seem counter-intuitive given Matlab's reputation for slowness with multiple layers of loops, as it is usually believed that Matlab performs faster with

[1]Using the following algorithm and $N_b = 400$, steps 1 to 3 in 1 account for 74% of the total runtime for the while-loop. Higher N_b values result in an even greater computational burden for these steps.

[2]Readers interested in more efficient max searching may exploit the monotonicity feature of the policy function, for example, the method in Gordon and Qiu (2018). However, parallel computation with this method is challenging due to data dependency across loops.

vectorization. In Appendix A, I present alternative implementations that exploit vector/matrix operations in native Matlab. However, as illustrated in Table 3, these alternatives take more time than the benchmark.

2.3 Setting Up Multi-threading on Matlab

In this section, I provide details on how to compile and execute parallelized code using Matlab Mex. At the hardware level, for CPU multithreading, it is ideal to have CPUs with multiple physical cores. For GPU multithreading, an NVIDIA GPU with a compute capability of 3.2 or higher is required.

2.3.1 Compiled CPU multithreading

The idea behind implementing Mex parfor is to allow Matlab to translate native Matlab code into C and/or C++ and parallelize the aforementioned for-loops using the OpenMP API. To enable Mex parfor, several Matlab toolboxes and software must be installed. Within Matlab, the Parallel Computing Toolbox and Matlab Coder are required. For a compiler that supports the OpenMP API, I choose MinGW version 10.3.0. Microsoft Visual Studio is another option, but its performance slightly lags behind MinGW on my experimental platform, especially when the workload is relatively light.

```
% generate mex parfor VFI function :
% ua is the utility after default -- u(y-L(y))
codegen -report solver -args {b,y,pdfy,ua} -o solver_Mex400
% call generated mex function :
[q,bp,vp,def,time] = solver_Mex400(b,y,pdfy,ua);
```

Listing 2: Generate a Mex parfor function for VFI and call it in a Matlab script.

Here are the procedures for using Mex parfor. First, I write a Matlab function file to implement the entire VFI while-loop. Second, I replace the outer 'for' loop in the third line of Listing 1 with 'parfor' to instruct the compiler to generate parallelized for-loops. There is no need to explicitly declare shared and private memory among threads. Third, I use the 'codegen' command to generate a VFI function in C/C++ based on the original Matlab function. In Listing 2, the VFI function is named "solver". The Mex parfor function, generated using the 'o' option, is named "solver_Mex400". The input matrices are declared with "-args {b,y,pdfy,ua}",

and a code generation report is requested using 'report' for diagnostic purposes. After successful code generation (which takes approximately 2 seconds in my case), I call "solverMex400" to obtain the solutions (q, $\mathcal{B}$, V^p, d) and runtime. Overall, the preparation of Mex parfor is straightforward.

2.3.2 Compiled GPU multithreading

The implementation of Mex CUDA requires additional software and procedures. At the software level, we need to install[3]:

(1) An NVIDIA GPU driver. For the RTX 3060 Laptop GPU, I installed the GeForce Game Ready Driver.

(2) The CUDA Toolkit, which enables CUDA programming on NVIDIA GPUs. I use version 10.2, the most recent version tested with the Matlab GPU Coder as of July 2024.

(3) A C/C++ compiler that supports the CUDA Toolkit on Matlab Coder.

With all these prerequisites installed, we can proceed with the coding. As with Mex parfor, I write the entire while-loop of VFI into a separate function file, as illustrated in Listing 3. The second line in the mock function instructs the code generator to map all operations with in this function into CUDA kernels. For complex for-loops within the function (lines 6 to 21), we should also add the pragma 'coder.gpu.kernel()' immediately before the loops. This directive tells the Matlab coder to convert the for-loops into CUDA kernels. Without the pragma "coder.gpu.kernel()", the generation of Mex CUDA code for the entire function will still work, but CUDA kernels for those for-loops may not be created, resulting in decreased performance. The "coder.gpu.constantMemory(y)" in line 6 declares vector y to be a read-only constant memory variable in the kernel, which improves memory management. Similar applications are found in lines 10 and 11.

```
1 function [q,bp,vp,def,time] = solver_mexGpu(b,y,pdfy,ua)
2 coder.gpu.kernelfun; % map the function into GPU kernels
3 vp = coder.nullcopy(zeros(ny,nb)) ; ... % pre-allocate memory
4 coder.gpu.kernel()  % map for-loops into GPU kernels
5 for iy = 1:ny
```

[3]See Supported and Compatible Compilers for reference and Installing Prerequisite Products for more supported CUDA Toolkit versions.

```
6   coder.gpu.constantMemory(y)  % map y to the constant memory
7   for ib = 1:nb
8     tmpmax = - Inf ;  bib = b(ib);
9     for i = 1:nb
10      coder.gpu.constantMemory(q);
11      coder.gpu.constantMemory(evp); % evp = betta*pdfy*vp
12      c1 =  b(i)*q(iy,i) - bib + y(iy);
13      if c1 <= 0; c1 = - Inf ;
14        else; c1 = (c1^(1-sigg)-1)/(1-sigg) + evp(iy,i);  end
15      if tmpmax < c1; tmpmax = c1 ; end
16    end
17    vp1(iy,ib) = tmpmax;
18  end
19 end
```

Listing 3: VFI function for Mex GPU generation. In a complete function these for-loops should be included into a while-loop. The complete function can be found in the appendix.

The instructions illustrated in Listing 4 generate Mex CUDA kernels within the function 'solver_mexGpu', thereby exploiting GPU multithreading. The use of 'codegen' function is similar to the Mex parfor case, but I pre-allocate input matrices/vectors into GPU memory (lines 3-6) and specifically use the first line to indicate that the compiled function should be CUDA Mex. The generated Mex CUDA function, 'solver_mexGpu400', is then called within Matlab. It is important to note that during compilation, the compiler checks the source code for structural errors before producing the object code. As a result, the created Mex CUDA function is only executable with the arguments specified during compilation (indicated with "-args {b,y,pdfy,ua}" in the 8th line). If users make any changes to the arguments and call the Mex CUDA function without recompiling, they will receive an error message. The compilation of Mex CUDA takes approximately 8 seconds.

```
1 cfg = coder.gpuConfig('mex');  %Specify configuration
2 cfg.GenerateReport = true;
3 b = gpuArray(b);  % Pre-allocate matrices into GPU memory
4 y = gpuArray(y);
5 pdfy = gpuArray(pdfy);
6 ua = gpuArray(ua);
7 % Transform the Matlab function into Mex CUDA function
```

```
codegen -config cfg solver_mexGpu -args {b,y,pdfy,ua} -o
    solver_mexGpu400
% call generated Mex CUDA function
[q,bp,vp,def,time] = solver_mexGpu400(b,y,pdfy,ua);
```

Listing 4: Generate a Mex CUDA function for VFI and call it in a Matlab script.

2.4 Efficiency Benchmarking: Multi-threading with Fortran

To benchmark the efficiency of Mex Parfor and Mex CUDA, I rewrote the Matlab code in Fortran, employing OpenMP with the GFortran compiler for CPU-based parallelization (Fortran omp) and OpenACC with the NvFortran compiler for GPU acceleration (Fortran acc). Fortran is chosen due to its longstanding reputation for efficient computation, its relative ease of use compared to C/C++ and CUDA, and its widespread application in solving nonlinear macroeconomic models, including sovereign default models (Arellano, 2008; Chatterjee and Eyigungor, 2012). In this section, I discuss the CPU and GPU-based parallelization in Fortran.

```
use omp_lib
!$omp parallel do default(shared) private(iy,ib,i,tmpmax,c1)
do iy = 1, ny
  do ib = 1, nb
    tmpmax = - huge(0.)
    do i = 1, nb
      c1 = b(i)*q(iy,i) - b(ib) + y(iy,1)
      if (c1 <= 0) then; c1 = tmpmax
        else; c1 = (c1**(1.0-sigg)-1.0)/(1.0-sigg)+evp(iy,i); end if
      if (tmpmax < c1) then; tmpmax = c1; end if
    end do
    vp1(iy,ib) = tmpmax
  end do
end do
!$omp end parallel do
gfortran -O3 -ffast-math -fopenmp -o omp400 omp.f95
```

Listing 5: Fortran codes (using OpenMP) for the calculation of V^p.

Listing 5 illustrates the Fortran code used to multi-thread the for-loops. the directive instructs the compiler to use OpenMP and defines the shared and private memory for each individual thread. , which contains the code to solve the sovereign default model, I use the compilation instructions listed in line 16. These instructions include options for the highest level of optimization, i.e., '-O3', and aggressive transformations concerning floating-point arithmetic to improve performance, i.e., '-ffast-math'.

```
1  use openacc
2  !$acc data copyin(pdfy, b, y, ua)
3  do while (diff > tol .and. its < 1000)
4    !$acc kernels
5    do iy = 1, ny
6      do ib = 1, nb
7        tmpmax = - huge(0.)
8        do i = 1, nb; ...; end do
9        vp1(iy,ib) = tmpmax
10     end do
11   end do
12   !$acc end kernels
13 end do
14 !$acc end data
15 nvfortran -acc -fast -O3 -gpu=managed -gpu=cc86 -o acc400 acc.f95
```

Listing 6: Fortran codes (using OpenACC) for the calculation of V^p.

Listing 5 illustrate the Fortran code incorporating GPU accelerate. The second line copies data into GPU memory, analogous to how the 'gpuArray' function operates in the Mex CUDA case. Together with the '!$acc end data' directive in line 15, this data copy ensures that all calculations within the while-loop are executed on the GPU. I use '!$acc kernels' and '!$acc end kernels' to instruct the compiler to generate GPU kernels for accelerating the for-loops, similar to the 'coder.gpu.kernel()' pragma in Mex CUDA. The compilation instructions are specified in line 16. I use '-gpu=managed' to enable simplified memory management for data transfer between the CPU and GPU. The '-gpu=cc86' option directs the compiler to optimize machine code for a target GPU with compute capability 8.6, which corresponds to NVIDIA's Ampere architecture for my host GPU.

3 Results and Discussions

3.1 Calibration and Platform

In the quantitative implementation, the utility function follows the standard constant relative risk aversion (CRRA) form:

$$u(c) = \frac{c^{1-\sigma} - 1}{1 - \sigma} \tag{6}$$

where the risk aversion parameter σ is set to 2. For other aspects of the calibration, the discount factor is $\beta = 0.90$, and the risk-free interest rate is $r^* = 0.01$. The endowment process has $\rho_y = 0.93165$ and $\sigma^y = 0.03696$. The probability transition matrix for the endowment, $\mathbb{E}y'|y$, is constructed using the method of Tauchen (1986), with 35 equally spaced grid points spanning 4.2 standard deviations around the unconditional mean. The endowment loss is given by $L(y) = \max\left\{0, \kappa 1 y + \kappa_2 y^2\right\}$, with $\kappa_1 = -0.35$ and $\kappa_2 = 0.44$. The probability of re-entry is $\mu = 0.0385$. Debt outstanding, b, is within the range $\mathbb{B} \in [0, 2]$, and the debt policy shares the same discretization. The while-loop for value function iteration (VFI) is terminated when $\Delta \leq 10^{-7}$.

Different numbers of grid points, N_b, for the endogenous state (debt) are tested: 400, 1500, 3000, and 6000. As many quantitative researchers on sovereign default models have argued, using 400 grid points seems satisfactory (Hatchondo et al., 2010, Uribe and Schmitt-Grohé, 2017) for a single endogenous state. However, when two endogenous states are incorporated, the total number of grid points can be very large. For example, using 60 grids for debt and 60 for capital results in $60 \times 60 = 3600$ grids in total[4]. We need to search throughout these 3600 grid points to locate the best policy combination that maximizes the sovereign's welfare. Therefore, exploring a large N_b in this single endogenous variable model provides valuable insights into the computational challenges and feasibility when dealing with multiple endogenous state variables.

The experiments were conducted on a Dell laptop equipped with an Intel Core i7-11800H CPU, featuring 8 physical cores. The system's Dynamic Random Access Memory (DRAM) is 16 GB, operating at 3200 MHz. The laptop also includes an NVIDIA GeForce RTX 3060 (Laptop version) GPU with 6 GB of dedicated memory. All Matlab/Mex programs were executed on

[4]Park (2017) discretizes debt and capital state spaces into 200 and 800 grid points, respectively. He parallelizes the computation on 30 CPU cores to manage the substantial workload. Arellano et al. (2018) use 72 equally spaced grids for capital and debt, resulting in 5184 coordinate points for policy choices. Na et al. (2018) uses 200 grids for debt and 150 grids for lagged real wage as two endogenous state variables, and solving that model requires 189,650 seconds using an Intel Xeon 3.5 GHz CPU and 256GB RAM.

Matlab 2023a and the corresponding Parallel Computation Toolbox and Matlab Coder, running on the Windows 11 operating system. The compiler for Mex parfor is TDM GCC 10.3.0, while Microsoft Visual Studio 2019 and CUDA Toolkit v10.2 were used for Mex CUDA compilation. For Fortran programs, the experiments were run on the Ubuntu 24.04 LTS (Long Term Support) system on the same laptop. The compilers used were GFortran version 12.2.0 for general Fortran compilation and NVIDIA HPC SDK version 24.5 for NvFortran[5].

3.2 Speed Comparison

In this section, I compare the time required to complete the entire VFI while-loop (illustrated in Listing 1) using different programming languages and workloads, as shown in Table 1. The term 'Matlab serial' refers to the non-parallelized native Matlab code[6]. Using Matlab serial as a benchmark, it is evident that compiled and parallelized Mex codes offer substantial speedups. 'Mex serial' denotes generated and compiled C/C++ code from the original Matlab script without multithreading acceleration. Mex serial achieves a 3-4 times speedup over Matlab serial, a factor similarly reported in Aruoba and Fernández-Villaverde (2015), demonstrating the efficiency advantage of compiled programs over interpreted ones. The speedup from Mex serial slightly lags behind the Fortran counterpart, 'Fortran serial'. However, as the workload increases, indicated by a larger N_b, their performance becomes more comparable.

For CPU-based multithreading, Mex parfor achieves a significant 20-fold acceleration over Matlab serial. Given that the computer has only 8 CPU cores, this 4-6 times performance advantage over Mex serial indicates that Mex parfor efficiently utilizes the multithreading potential within the physical limits of the hardware. The Fortran multithreading counterpart, Fortran omp, offers performance similar to Mex parfor[7]. Initially, at $N_b = 400$, the speedup is 13-fold. It then increases, peaking at 22-fold, before deteriorating to 14-fold at $N_b = 6000$.

GPU-based multithreading demonstrates superior performance on my platform, particularly

[5]GFortran exhibits slightly better performance on Linux compared to Windows. NvFortran is well-supported on Linux but is currently not feasible on Windows. Matlab and Mex programs were executed on Windows to accommodate the majority of Matlab users.

[6]Writing the while-loop in a separate Matlab function file allows Matlab to compile the function at runtime using just-in-time (JIT) compilation, providing a moderate speedup.

[7]Fernández-Villaverde and Valencia (2018) reported a speedup of 3 times with C++ and OpenMP when all 8 cores were employed. The higher acceleration in this paper may stem from differences in CPU architecture, DRAM, the model, and the algorithm. Other languages like Rcpp of R, native Matlab, and Python provide less than four times speedup, while their R and Julia counterparts achieved less than two times speedup.

Table 1: Time spent (seconds) and speedup over Matlab serial in completing VFI for the sovereign default model: Comparison between number of grids, languages, and methods

	Number of grid points for debt			
Methods	$N_b = 400$	$N_b = 1500$	$N_b = 3000$	$N_b = 6000$
Time spent in seconds				
Matlab serial	7.404	111.968	427.364	2003.719
Mex serial	2.216	32.342	107.793	553.382
Fortran serial	1.482	23.673	90.801	445.914
Mex parfor	0.582	5.203	19.356	141.181
Fortran omp	0.381	4.404	17.607	152.989
Mex CUDA	0.297	2.689	8.895	37.403
Fortran acc	0.295	3.064	11.161	47.430
Mex CUDA FP32	0.134	0.383	0.863	3.003
Fortran acc FP32	0.097	0.517	1.241	4.128
Speedup over Matlab serial				
Mex serial	3.34	3.46	3.95	3.62
Fortran serial	5.00	4.73	4.71	4.49
Mex parfor	12.72	21.52	22.08	14.19
Fortran omp	19.43	25.43	24.27	13.10
Mex CUDA	24.94	41.64	48.04	53.57
Fortran acc	25.08	36.55	38.29	42.25
Mex CUDA FP32	55.30	292.25	495.19	667.16
Fortran acc FP32	76.59	216.55	344.46	485.35

Notes: Fortran omp denotes the combination of GFortran with OpenMP. Fortran acc denotes the combination of OpenACC with NvFortran. FP32 denotes the calculation is based on single-precision data. Without specification, computation is based on double precision, which is the default for Matlab.

with higher workloads. Mex CUDA achieves a 25-fold speedup over Matlab serial at the lowest workload. As N_b increases, the advantage of GPU computation becomes more pronounced. At $N_b = 1500$, the acceleration reaches 41-fold, and at the highest $N_b = 6000$, it reaches 53-fold, completing VFI in just half a minute compared to half an hour with Matlab serial. Compared to Mex parfor, Mex CUDA is still more than 4 times faster at the heaviest workload. All computations so far have used double-precision floating-point format (FP64), the default for Matlab. If the precision requirement is relaxed to single-precision (FP32), the speedup further increases, reaching 55-fold at $N_b = 400$ and an impressive 660-fold at $N_b = 6000$[8]. These results underline the remarkable advantage of GPU-based parallelization in macroeconomic computation.

Table 1 also reveals that the Fortran GPU counterpart, Fortran acc, offers comparable yet slightly slower speeds than Mex CUDA. This disparity could stem from differences in the operating systems (Windows for Mex CUDA and Linux for Fortran acc) and compilation. During compilation, both the Mex CUDA and NvFortran compilers translate the original Matlab or Fortran codes into CUDA codes, leveraging user-added pragmas and directives. The similar performance levels indicate that Matlab's GPU Coder effectively generates optimized CUDA codes, achieving performance comparable to that of OpenACC in NvFortran. Using the 'cudafor' library (CUDA Fortran), the NVIDIA HPC SDK compiler provides additional fine-tuning options, such as writing explicit GPU kernels and manually managing GPU memory. This makes it more akin to using CUDA and potentially faster. In contrast, the combination of Fortran, OpenACC, and 'acc kernels'[9] is a pragma-based approach requiring minimal changes to Fortran codes, demanding less understanding of GPU architecture and reduced learning efforts. However, this convenience might come at the cost of performance, potentially lagging behind more hands-on approaches like CUDA and CUDA Fortran.

3.3 Performance and Architecture

In the preceding sections, we observed the speedup of compiled multithreading varies with the number of grid points. For Mex parfor and Fortran omp, the speedup peaks at around

[8]Using FP32 with native Matlab code is slower than FP64. Fortran serial and omp using FP32 achieve a 2-fold speed improvement over FP64. For this sovereign default model, using FP32 (either on Fortran or Mex) only slightly changes the solution. For example, with $N_b = 6000$, the Euclidean norm between q in FP32 and FP64 is $1.4e^{-5}$; with $N_b = 400$, it is $3.63e^{-6}$. Simulation results based on these solutions show negligible differences.

[9]Readers interested in further fine-tuning in the Fortran OpenACC environment are referred to the '!$acc parallel' approach discussed in Abramkina et al. (2023).

$N_b = 3000$, while the advantage of Mex CUDA and Fortran acc grows with the data size. This distinction highlights a trade-off: given efficient compilation, CPU-based multithreading tends to perform better with smaller workloads, while GPU-based multithreading is more advantageous for larger workloads. One reason for this is that CPU-based multithreading accelerates the for-loops defined in lines 3-15 in Listing 1, but not explicitly for the rest of the vectorized matrix arithmetic such as calculating $\mathbb{E}V^p$, V^d, q, and so on. While it is possible to write explicit loops to replace vectorization in these cases, doing so usually drags down performance unless the matrices are very large. However, GPU compilers like NvFortran and Mex CUDA can recognize and translate vectorized matrix operations into CUDA kernels, thereby achieving significant acceleration.

Another significant factor influencing performance is the hardware architecture. GPUs are specifically designed for parallelization, featuring a large number of cores. In my GPU, there are 3840 CUDA cores supporting 480 threads that can work simultaneously. These threads are organized into 30 Streaming Multiprocessors (SMs), each capable of performing independent calculations. In contrast, CPU multithreading on our laptop utilizes far fewer threads. The CPU has only eight physical cores, and the Mex parfor approach uses one core per thread. The GPU's ability to employ a high number of threads stems from its specialized cores designed for simple, specific tasks. This specialization allows GPU cores to be smaller and more numerous, facilitating extensive parallelism. On the other hand, CPU cores are designed for more complex and generalized tasks, which makes each core larger and causes inter-core communication to be relatively slow.

In addition to the difference in the number of threads and cores, another significant factor is memory bandwidth and latency, as illustrated in Figure 1. The GPU in our setup has a memory bandwidth of 295 gigabytes per second (GB/s) and a latency of 145 nanoseconds (ns). To fully utilize its floating-point operations per second (FLOPS) capacity, the GPU needs to fetch 295 GB/s $\times$ 145ns $= 42,775$ bytes of data per memory access, known as the peak bytes per latency[10]. CUDA programming aims to keep the GPU memory oversubscribed, ensuring that threads remain busy in calculation pipelines and thereby maximizing the use of its FLOPS capacity (Jones, 2021; Jones, 2022). In contrast, the CPU has a lower memory bandwidth of 39.4

[10]This data is sourced from the AIDA 64 test on a AMD 4700S desktop kit, which uses GDDR6 for it main memory, the same type as our GPU. For consistency, the FLOPS, bandwidth, and latency are all tested on AIDA 64. For reference, see the AMD 4700S Review: Defective PlayStation 5 Chips Resurrected.

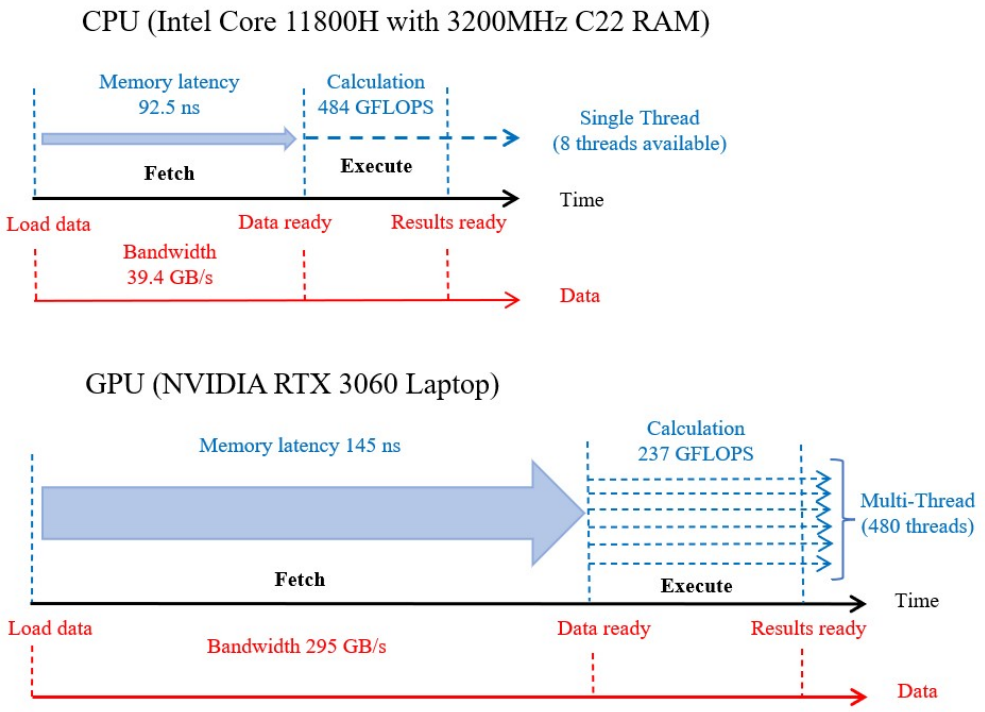

Figure 1: An illustration of memory latency in CPU and GPU. The GFLOPSs refer to one billion floating-point operations per second in double precision. FLOPS and Memory data is tested via the AIDA64 GPGPU Benchmark and AIDA64 Cache & Memory Benchmark.

GB/s and a latency of 92.5 ns, resulting in a significantly lower peak bytes per latency of 3,645 bytes. To process the 42,775 bytes of data that the GPU can handle in a single fetch-execute cycle, the CPU would need approximately 11.7 cycles. Given the comparable FLOPS capacities between our CPU and GPU, the CPU's limited memory bandwidth makes it more susceptible to becoming memory-bound in multithreading scenarios.

The efficiency advantage of using single-precision data is also deeply rooted in the architecture of GPUs. For consumer-level GPUs, the theoretical floating-point operations per second (FLOPS) at single precision (FP32) are significantly higher compared to double precision (FP64). This is because each Streaming Multiprocessor (SM) contains 128 FP32 units but only two FP64 units, resulting in a FLOPS capacity that is 64 times greater for FP32 compared to FP64. Additionally, each FP64 data element requires 8 bytes, while FP32 data elements require only 4 bytes, making FP32 data transfers twice as efficient in terms of memory bandwidth. Due to efficient memory management of the CUDA programming, Mex CUDA achieves approximately 12 times when using FP32 compared to FP64. This substantial performance gap is a consequence of the GPU's architecture being optimized for gaming and graphics work-

loads, which predominantly utilize FP32. In contrast, professional GPUs designed for scientific computation, such as the NVIDIA A100, feature FP64 units in a 1:2 ratio compared to FP32 units. This architectural distinction allows professional GPUs to better handle tasks requiring double-precision computations.

4 Sensitivity analysis: A stochastic growth model

In this section, I use a stochastic growth model to illustrate the performance improvements achieved through compiled multithreading in Matlab. This model serves as a common example in macroeconomic computations and provides a context for evaluating algorithmic efficiency. The objective is to solve the dynamic programming problem to find the value function V and optimal capital policy $\mathcal{K}$. The problem can be formulated as:

$$V(z,k) = \max_{c\geq 0,\, k'} u(c) + \beta \mathbb{E}_{z'|z} V(z',k') \tag{7}$$

where β is the subjective discount factor, and $c = e^z k^\alpha + (1-\delta)k - k'$, where c, k and z denotes consumption, capital stock and productivity, respectively. δ is the depreciation rate. The stochastic productivity z follows an AR(1) process: $z' = \rho_z z + \varepsilon^z$ with $\varepsilon^z \sim i.i.N(0, \sigma^z)$.

For calibration, $\beta = 0.95$, $\sigma = 2$, $\alpha = 0.33$, $\delta = 0.1$, $\rho_z = 0.9$ and $\sigma^z = 0.01$. The stochastic variable z is discretized using a probability transition matrix with 35 equally spaced grid points, denoted as $\mathbb{Z}$. The capital state and policy grids are also discretized, with N_k taking values of 400, 1500, 3000, and 6000 grid points. These grids span between 25% and 400% deviations from the steady-state value. The utility function is defined as Equation 6. We solve this stochastic growth model using the same VFI algorithm employed for the sovereign default model, though the stochastic growth model involves a smaller computational workload and reduced complexity in arithmetic operations.

As shown in Table 2, both Mex parfor and Mex CUDA demonstrate substantial performance improvements over the serial method using Matlab code. The speedups achieved with Mex parfor (ranging from 13 to 25 times) and Fortran omp (ranging from 27 to 33 times) surpass those observed in the sovereign default model (12 to 22 times for Mex parfor; 19 to 25 times for Fortran omp). This indicates that CPU-based parallelization methods tend to perform better with smaller workloads. In contrast, the performance of GPU-based methods shows a decline.

Table 2: Time spent (seconds) and speedup over Matlab serial in completing VFI for the stochastic growth model: Comparison between number of grids, languages, and methods

	Number of grid points for debt			
Methods	$N_b = 400$	$N_b = 1500$	$N_b = 3000$	$N_b = 6000$
Time spent in seconds				
Matlab serial	6.133	86.532	332.888	1318.434
Mex serial	2.139	29.889	81.794	359.362
Fortran serial	1.060	15.063	60.694	243.917
Mex parfor	0.449	3.423	12.984	53.208
Fortran omp	0.224	2.571	10.197	41.103
Mex CUDA	0.273	3.106	9.679	34.930
Fortran acc	0.265	2.348	8.508	32.679
Mex CUDA FP32	0.065	0.247	0.638	2.326
Fortran acc FP32	0.062	0.254	0.605	1.875
Speedup over Matlab serial				
Mex serial	2.87	2.90	4.07	3.67
Fortran serial	5.78	5.74	5.48	5.41
Mex parfor	13.67	25.28	25.64	24.78
Fortran omp	27.43	33.66	32.65	32.08
Mex CUDA	22.50	27.86	34.39	37.74
Fortran acc	23.16	36.85	39.13	40.34
Mex CUDA FP32	94.25	349.98	521.78	566.87
Fortran acc FP32	99.25	340.79	550.57	703.10

Notes: Fortran omp denotes using GFortran and OpenMP to parallelize for-loops in the VFI. Fortran acc denotes using OpenACC to accelerate NvFortran codes.

Mex CUDA provides speedups of 22 to 37 times, whereas the sovereign default counterparts achieve 25 to 53 times. Similarly, NvFortran FP64 exhibits slightly lower acceleration compared to Matlab serial. However, it suggests that switching to single-precision (FP32) data is more beneficial for simpler models. For the stochastic growth model, Mex CUDA FP32 achieves a speedup of 4 to 15 times over FP64, and NvFortran FP32 offers a comparable 4 to 17 times acceleration. In the more complex sovereign default model, the speedup decreases to 2 to 12 times with Mex CUDA and 3 to 11 times with NvFortran.

5 Concluding remarks

In this paper, utilizing the convenient Matlab executable (Mex) approach, I have explored CPU and GPU parallelization to accelerate the grid-based value function iteration solving of a sovereign default model. The results demonstrate significant speed improvements over conventional serial Matlab execution. Performance-wise, the Mex approach is comparable to the renowned GFortran (for CPU) and NvFortran (for GPU) approaches. Importantly, utilizing Mex does not necessitate familiarity with Fortran, CUDA, or C/C++, as Matlab Coder, with the addition of appropriate pragmas/directives, effectively translates native Matlab code into compiled C/C++ or CUDA code. This makes it accessible to a broad spectrum of economic researchers who are accustomed to Matlab and appreciate its user-friendly environment for data analysis. I also find CPU multithreading performs better with smaller workloads, while GPU excels with larger datasets and more matrix arithmetic operations Notably, using single precision with Mex CUDA can provide an additional ten-fold acceleration over double precision when appropriate for the computational context.

These findings suggest that Mex-based parallelization can be broadly beneficial in computation-intensive economic research, significantly enhancing computational efficiency with a relatively small investment of time and effort. However, while pragma-based compiled parallelization is straightforward to implement, it lacks the fine-tuning and optimization potential that C, CUDA, and CUDA Fortran can offer. Thus, for those interested in achieving further performance gains, I recommend studying these lower-level languages and gaining a deeper understanding of hardware architectures.

References

Abramkina, O., Dubois, R., and Véry, T. (2023). Openacc for GPU: an introduction. Technical report, Institute for Development and Resources in Intensive Scientific Computing.

Aguiar, M. and Gopinath, G. (2006). Defaultable debt, interest rates and the current account. *Journal of international Economics*, 69(1):64–83.

Aldrich, E. M. (2014). Gpu computing in economics. In *Handbook of Computational Economics*, volume 3, pages 557–598. Elsevier.

Aldrich, E. M., Fernández-Villaverde, J., Gallant, A. R., and Rubio-Ramírez, J. F. (2011). Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors. *Journal of Economic Dynamics and Control*, 35(3):386–393.

Arellano, C. (2008). Default risk and income fluctuations in emerging economies. *American Economic Review*, 98(3):690–712.

Arellano, C., Bai, Y., and Mihalache, G. (2018). Default risk, sectoral reallocation, and persistent recessions. *Journal of International Economics*, 112:182–199.

Arellano, C., Maliar, L., Maliar, S., and Tsyrennikov, V. (2016). Envelope condition method with an application to default risk models. *Journal of Economic Dynamics and Control*, 69:436–459.

Aruoba, S. B. and Fernández-Villaverde, J. (2015). A comparison of programming languages in macroeconomics. *Journal of Economic Dynamics and Control*, 58:265–273.

Chatterjee, S. and Eyigungor, B. (2012). Maturity, indebtedness, and default risk. *American Economic Review*, 102(6):2674–99.

Coleman, C., Lyon, S., Maliar, L., and Maliar, S. (2021). Matlab, python, julia: What to choose in economics? *Computational Economics*, 58(4):1263–1288.

Deng, M., Guerron-Quintana, P. A., and Tseng, L. (2022). Parallel computation of sovereign default models. *Computational Economics*, pages 1–39.

Fernández-Villaverde, J. and Valencia, D. Z. (2018). A practical guide to parallelization in economics. Technical report, National Bureau of Economic Research.

Gordon, G. and Qiu, S. (2018). A divide and conquer algorithm for exploiting policy function monotonicity. *Quantitative Economics*, 9(2):521–540.

Hatchondo, J. C., Martinez, L., and Sapriza, H. (2010). Quantitative properties of sovereign default models: solution methods matter. *Review of Economic dynamics*, 13(4):919–933.

Jones, S. (2021). How gpu computing works. In *GPU Technology Conference Digital April*. NVIDIA.

Jones, S. (2022). How cuda programming works. In *GPU Technology Conference Digital Spring*. NVIDIA.

Maliar, L. and Maliar, S. (2014). Numerical methods for large-scale dynamic economic models. In *Handbook of computational economics*, volume 3, pages 325–477. Elsevier.

Morozov, S. and Mathur, S. (2012). Massively parallel computation using graphics processors with application to optimal experimentation in dynamic control. *Computational Economics*, 40:151–182.

Na, S., Schmitt-Grohé, S., Uribe, M., and Yue, V. (2018). The twin ds: Optimal default and devaluation. *American Economic Review*, 108(7):1773–1819.

Park, J. (2017). Sovereign default and capital accumulation. *Journal of International Economics*, 106:119–133.

Richter, A. W., Throckmorton, N. A., and Walker, T. B. (2014). Accuracy, speed and robustness of policy function iteration. *Computational Economics*, 44:445–476.

Schmitt-Grohé, S. and Uribe, M. (2004). Solving dynamic general equilibrium models using a second-order approximation to the policy function. *Journal of economic dynamics and control*, 28(4):755–775.

Sims, C. A. (2002). Solving linear rational expectations models. *Computational economics*, 20(1-2):1.

Tauchen, G. (1986). Finite state markov-chain approximations to univariate and vector autoregressions. *Economics Letters*, 20(2):177–181.

Uribe, M. and Schmitt-Grohé, S. (2017). *Open Economy Macroeconomics*. Princeton University Press.

Appendices

A Alternative methods to calculate the value of repayment in the sovereign default model

In this section I present three alternative methods to calculate V^p for the sovereign default model, with native Matlab execution. The first method (listed at 7) is the full vectorization (FV) method that is used in Aguiar and Gopinath (2006) etc. In this method, the vectors representing grid points for endowment $y(y,1)$, outstanding debts $b(1,b)$, debt issuance $b'(1,b')$, and price $q(y,b')$ are reformulated to be $y(y\times b,b)$, $b(y\times b,b)$, $b'(y\times b,b)$ (the 'bp' matrix in list 7), and $q(y\times b,b')$ matrices where each row index of them corresponds to a combination of grid points of two state variables, $\{y_i\}$ and $\{b_i\}$, and the column index corresponds to grid points of candidate debt policies.

```
c =  bp.*repmat(q,[nb 1]) - b + y ;      % bp is debt issuance
u = (c.^(1-sigg) - 1) / (1 - sigg);
u(c<=0) = - Inf;
% vp1 is (ny,nb) matrix
vp1(:) = max(u + repmat(betta*pdfy*vp,nb,1), [], 2);
```

Listing 7: Matlab codes for the FV method to calculate V^p

The second method (called "Looping Over All States" (LOAS), illustrated in 8), employs two layers of nested loops to sequentially visit all the grid points of states (y_i and b_i). For each coordinate (y_i,b_i), it evaluates an $(1,N_b)$ vector of welfare, which is $u(c(1,b'))+\beta\mathbb{E}V^p(1,b')$, to obtain the updated value function of size $(1,1)$ by finding the welfare maximizing column.

```
evp = betta*pdfy*vp ;
for iy = 1:ny
  for ib = 1:nb
    c =  bp.*q(iy,:) - bp(ib) + y(iy) ;   % bp is (1,nb) vector
    u = (c.^(1-sigg) -1) / (1-sigg) ;    % c and u are (1,nb) vector
    u(c<=0) = - Inf ;    % set utility with c<=0 to be very small
    vp1(iy,id)  = max(u + evp(iy,:)) ;
  end
```

```
end
```

Listing 8: Matlab codes for the LOAS method to calculate V^p

The third method is named as the "Looping Over Exogenous States" (LOES) algorithm, as illustrated in List 9. It uses a single "for" loop to serially visit all N_y grids for the endowment $\{y_i\}$, while in each of these iterations it evaluates an (N_b, N_b) welfare matrix, i.e. $u(c(b,b')) + \beta \mathbb{E} V^p(b,b')$, to find the value function V^p as an $(N_b, 1)$ vector.

```
for iy = 1:ny
    c =  bp.*q(iy,:) - b + y(iy) ;   % bp is (1,nb), b is (nb,1)
    u = (c.^(1-sigg) -1)/(1-sigg) ;  % c and u are (nb,nb) matrices
    u(c <= 0) = - Inf ;
    vp1(iy,:) = max(utry + betta*pdfy(iy,:)*vp,[],2) ;
end
```

Listing 9: Matlab codes for the LOES method to calculate V^p

Table 3: Time spent (in seconds) for alternative methods with Matlab serial codes to complete the VFI while-loop for the sovereign default model

N_b	Method	Time	Speedup
400	FV	16.545	0.37
	LOAS	18.157	0.34
	LOES	7.744	0.79
1500	FV	237.354	0.37
	LOAS	200.116	0.43
	LOES	132.479	0.65
3000	FV	N.A.	N.A.
	LOAS	597.215	0.56
	LOES	536.505	0.62
6000	FV	N.A.	N.A.
	LOAS	2326.200	0.57
	LOES	2427.567	0.54

Notes: "Speedup" in this table refers to the time spent with the benchmark method over corresponding alternative methods.

As mentioned in the main text, the benchmark three-for-loop method outperforms all these alternative methods in Matlab serial execution, as proven by Table 3. For Mex parfor and Mex CUDA, these alternative methods deliver similar speed as the benchmark counterparts. The FV results for $N_b = 3000$ and $N_b = 6000$ are not available because in these cases the computer runs out of memory and can not proceed.