Efficient Parallelization of Nonlinear Macroeconomic Models: A MATLAB Executable Approach

Alessandro Di Nola[*]
University of Birmingham

Liang Shi[†]
University of Essex

January 2026

Abstract

Solving nonlinear macroeconomic models requires substantial programming effort and computational time, often constraining research and application. Using state-of-the-art sovereign default and heterogeneous-agent models, this paper evaluates how parallelized computation with MATLAB Executables (MEX) can alleviate these constraints. This hybrid approach translates MATLAB code into C (MEX-parfor) and CUDA (MEX-CUDA) executables that run seamlessly within the MATLAB environment. Compared with MATLAB's native parallel computation, MEX-parfor achieves speedups of up to 500-fold, while MEX-CUDA delivers gains exceeding 1,000-fold, comparable to other highly optimized languages. Moreover, the method can be readily deployed on cloud platforms such as Amazon Web Services (AWS), enabling solving high-dimensional nonlinear models with minimal additional development effort.

Keywords: Parallelization, MATLAB MEX, GPU computation, Sovereign default, Heterogeneous agents, Dynamic programming

JEL Codes: C63, C88, F41

[*]Department of Economics, University of Birmingham, Edgbaston, Birmingham, B15 2TT, United Kingdom. E-mail: a.dinola@bham.ac.uk

[†]Institute for Social and Economic Research (ISER), University of Essex, Wivenhoe Park, CO4 3SQ, Colchester, United Kingdom. E-mail: shiliang369econ@gmail.com

Introduction

Nonlinear economic models often rely on iterative procedures, such as value function iteration (VFI), to obtain solutions, but their computational burden has long constrained progress in both research and application. Examples include models in quantitative macroeconomics with heterogeneous agents, sovereign default models, structural models in labor economics and industrial organization. To loosen this constraint, economists have increasingly turned to modern hardware with high parallelization potential and to software tools that can efficiently exploit this potential. Compiled languages such as C, Fortran, and CUDA are promising candidates, but they require substantial development time and specialized expertise. In contrast, scripted languages such as MATLAB and Python are popular for their ease of development and deployment, but they lack execution efficiency, especially in parallelized settings.

Recent advances in programming have bridged the gap between rapid development and fast execution, leading to a new generation of hybrid approaches that combine the advantages of scripted and compiled languages (Aruoba and Fernández-Villaverde, 2015). For example, machine learning frameworks such as TensorFlow and PyTorch achieve high execution speed through the integration of compilation-based kernels (Duarte et al., 2020). Just-in-time (JIT) compilation has also driven progress in Julia (Deng et al., 2022) and MATLAB. MATLAB Executables (MEX) and Rcpp enable users to compile native code and execute it within their familiar integrated development environments (IDEs). Such hybrid methods are highly promising for alleviating computational bottlenecks across a broad range of economic applications.

In this paper, we propose a hybrid method that combines the advantages of scripted and compiled languages. Specifically, we examine parallelization through MATLAB Code Generation, utilizing both central processing units (CPUs) and graphics processing units (GPUs) via MEX. This method translates native MATLAB functions into C or CUDA code, compiles them into executables, and runs them within the MATLAB environment. On a standard personal computer (PC), we demonstrate that this approach can accelerate the solution of nonlinear economic models by several hundred-fold while preserving the convenience of MATLAB-based development. Moreover, the same MEX framework can be easily deployed on computing clusters and cloud platforms, as we illustrate using Amazon Web Services (AWS). This method has not been previously documented in the economic literature and, to the best of our knowledge, has not been widely applied in the field.

We illustrate the power of MEX-based parallelization through two representative nonlinear macroeconomic models that require computationally demanding VFI routines: a sovereign default model and a heterogeneous agents model. By illustrating this method with two widely used models in macroeconomics and making the code available, we allow researchers to apply the method for their own computational tasks in a wide array of applications.[1] To evaluate performance under realistic levels of complexity, we incorporate structural features that substantially increase computational cost. The sovereign default model includes long-maturity debt (Chatterjee and Eyigungor, 2012), debt renegotiation (Mihalache, 2020), and two exogenous shocks, making it considerably more complex than the canonical model of Arellano (2008). For the heterogeneous agents model, we adopt the framework of Buera and Shin (2013), characterized by multiple state variables and a discrete occupational choice that leads to a non-convex feasible set in the dynamic programming problem.

In these benchmark implementations, we employ a brute-force approach that evaluates all discretized policy grids. We then explore MEX performance under alternative algorithms, specifically, linear interpolation and the binary monotonicity method of Gordon and Qiu (2018). Because these methods involve higher memory overhead than brute force discretization, MEX multithreading on CPU (MEX-parfor) does not always outperform native MATLAB. However, MEX's GPU implementation (MEX-CUDA) continues to offer substantial speedups through efficient memory management, aided by the mature CUDA Toolkit developed by NVIDIA. Section 5 discusses how to adjust MATLAB code to generate efficient CUDA kernels that fully utilize the GPU while keeping development effort manageable. The pragma-based coding style in MEX is also considerably more convenient than the GPU programming grammar in Julia, which closely resembles CUDA syntax.

Our experiments further show that MEX-CUDA achieves execution performance comparable to that of the compiled language CUDA Fortran and the widely used GPU computing framework PyTorch. Finally, because MEX-CUDA operates primarily through loop-based structures, it allows a minimum GPU memory occupation, widest adaptivity to complex models and better speed comparing with vectorized methods.

It is worth highlighting that a GPU is not a general-purpose processor. It is designed

[1]Replication code is available in authors' GitHub repositories.

for massively parallel workloads, and its relative performance improves with computational intensity. Achieving high GPU performance requires both appropriate coding styles and efficient compiler support. Our experiments show that MEX-CUDA performs best when large workloads are divided into smaller tasks, each occupying minimal stack memory per thread. In this regard, MATLAB GPU Coder provides effective diagnostic tools and compiler directives (pragmas) to facilitate memory management and code optimization.

The remainder of the paper is organized as follows. Section 2 provides a brief overview of MEX-based parallelization. Section 3 presents experiments based on the sovereign default model, including detailed performance comparisons across implementation methods and algorithms. Section 4 applies MEX to a heterogeneous-agents model. Section 5 discusses code design for efficient MEX-CUDA kernel generation, and Section 6 concludes.

1 Literature review

This paper contributes to the computational economics literature on parallelization using CPUs and GPUs. Aldrich et al. (2011) report a 300-fold speedup using CUDA relative to a C++ serial CPU implementation when solving a stochastic growth model. Similarly, Morozov and Mathur (2012) analyze CUDA performance in dynamic models with imperfect information and find roughly a 15-fold improvement over single-threaded Fortran implementations. Aruoba and Fernández-Villaverde (2015) and Fernández-Villaverde and Valencia (2018) compare a wide range of programming languages from a quantitative economist's perspective and predict the increasing adoption of hybrid methods that combine ease of development with high computational efficiency. Other hybrid approaches include the use of machine learning frameworks such as TensorFlow and PyTorch (Duarte et al., 2020), which couple Python with highly optimized back-end CUDA kernels, achieving execution speeds comparable to compiled languages.

We add to this literature by exploring MEX as another hybrid method for applying parallelized computation to nonlinear macroeconomic models with modern structural complexity. Our sovereign default model extends the canonical one-period debt framework of Duarte et al. (2020) and Deng et al. (2022) by introducing a substantially more complex structure. The heterogeneous-agents model serves as a sensitivity analysis and demonstrates the applicability of MEX-based methods to another major area of quantitative macroeconomic research.

Previous work has examined MEX primarily as an interface for calling compiled languages. For instance, Richter et al. (2014) report an eightfold speedup in a real business cycle model and a 24-fold speedup in a New Keynesian model by executing Fortran multithreading routines within MATLAB. In contrast, our approach uses C/CUDA code generation and compilation directly from native MATLAB code. This feature makes MEX particularly valuable for the large community of economists and students working in scripted environments (Coleman et al., 2021), as it reduces both programming effort and execution time.

Kirkby (2017) develops the VFI Toolkit, which exploits GPU parallelization in MATLAB through vectorized commands such as `gpuArray`. His approach is complementary to ours, though our loop-based method is less memory intensive as it avoids preallocating large arrays and suits C/CUDA compilation better in execution speed.

Beyond GPUs, specialized hardware platforms such as Field-Programmable Gate Arrays (FPGAs) (Peri, 2020) and Application-Specific Integrated Circuits (ASICs) offer even greater acceleration potential, though they require substantially more development effort and technical expertise for practical use.

Finally, this study relates to a broader class of algorithms designed to accelerate the global solution of nonlinear macroeconomic models. Examples include the endogenous grid points method (Carroll, 2006), the envelope condition method (Arellano et al., 2016), and the fixed-point iteration method (Mendoza and Villalvazo, 2020). Although these algorithms are not examined here, they would likely benefit from MEX-based parallelization. We view this as a promising direction for future work and anticipate that adjustments to GPU-oriented coding would be valuable.

2 A MEX-based method for efficient parallelization

A MEX file bridges the MATLAB programming environment with functions written in compiled languages such as C, C++, or Fortran. Using the `codegen` command, MATLAB (GPU) Coder can automatically translate functions written in MATLAB language into C or CUDA code. These translated functions are compiled into executable files (MEX) that can be called directly from within MATLAB's integrated development environment (IDE) using MATLAB syntax. This approach usually helps accelerate computation with the convenient MATLAB-based development environment.

The performance gains are even more pronounced when parallelization is employed. MATLAB's native `parfor` (hereafter **MATLAB parfor**) distributes loop iterations in interpreted environments, which means each worker incurs interpretation overhead. This often creates a performance bottleneck, particularly for complex computational structures. As our results show, MATLAB parfor can often underperform the serial/vectorized execution of native MATLAB code. The compilation step in the MEX-based method, instead, removes most interpretation overhead and organizes execution more efficiently using third party APIs.

MEX-parfor and MEX-CUDA are straightforward to implement once the required software and hardware prerequisites are in place. The user needs only a MATLAB licence with the relevant toolboxes and a compatible C/C++ compiler; if MEX-CUDA is used, an NVIDIA graphics card is additionally required. These conditions are typically met on a standard personal computer. Appendix B lists the specific hardware and software configurations used for our benchmark examples.

As illustrated in Listing 1, programmers can isolate the most computationally intensive blocks (e.g. the value function iteration) into a standalone function (e.g., `solver.m`), add the appropriate pragmas (e.g., `parfor`, `coder.gpu.kernelfun`, to be discussed in Section 5), and use the `codegen` command to generate a MEX file. Argument types are specified using `-args {}`, the output name is assigned with `-o`, and the code-generation report is enabled via `cfg.GenerateReport = true` for diagnostics and refinement.[2]

2.1 Applying MEX on clouds

Beyond single-device parallelization on a CPU or GPU, MEX also supports distributed execution across clusters and cloud platforms such as Amazon Web Services (AWS), Microsoft Azure, or institutional high-performance computing (HPC) systems. This capability makes MEX methods practical for economists tackling large-scale nonlinear models. Therefore, in addition to experiments conducted on our PC, we also test MEX using cutting edge hardware on AWS for reference.

To build an environment with all MEX prerequisites on AWS, we launch an EC2 instance and install MATLAB with the required toolboxes, GPU drivers, and third-party compilers

[2]Users can define variable-size matrices as inputs using the `coder.typeof()` command. In the benchmark results reported in Tables 1 and 6, we include the execution time of MEX with variable-size inputs.

```
%% Example 1: Generate MEX-parfor VFI solver for the benchmark sovereign
   default model
cfg = coder.config('mex');  % Specify configuration
cfg.GenerateReport = true;  % Enable code generation report
% Write MATLAB function ‘solver’ into MEX file ‘solver_MEX’:
codegen -config cfg solver -args {z,m,b,pdf_joint,para} -o solver_mex
[vp,vd,q,bp,bpr,default,rr,totaltime,avgtime] = ...
    solver_mex(z,m,b,pdf_joint,para); % Call MEX function

%% Example 2: Generate MEX-CUDA VFI solver for the benchmark sovereign
   default model
cfg = coder.gpuConfig('mex');  % Specify configuration
cfg.GenerateReport = true;     % Enable code generation report
% Write MATLAB function ‘solver_gpu’ into MEX file ‘solver_gpu_mex’:
codegen -config cfg solver_cuda -args {z,m,b,pdf_joint,para} ...
    -o solver_gpu_mex
[vp,vd,q,bp,bpr,default,rr,totaltime,avgtime] = ...
    solver_cuda_mex(z,m,b,pdf_joint,para); % Call MEX function
```

Listing 1: MATLAB code that generates a MEX-parfor function to implement VFI and call it. Vectors z and m are vectors of exogenous shocks, b is the vector of debt grids, `pdf_joint` is the joint probability transition matrix for z and m shocks, para is a structure array that contains relevant parameters. Details of the model is in Section 3.

(See Online Appendix B for details.). Users can create a custom Amazon Machine Image (AMI) based on the environment of this instance, which can be reused to deploy additional instances. [3] While users can generate MEX files directly on AWS, they may also compile them locally and upload them to an AWS instance for execution. This approach eliminates the need to install MATLAB Coder or third-party compilers on AWS but forgoes the device-specific optimizations that code generation can provide.

3 Experiment 1: Sovereign default model

This section discusses the performance of the MEX approach in solving a complex sovereign default model, a representative class of nonlinear macroeconomic models that are computationally intensive. The sovereign default model builds on the long-maturity debt structure in (Chatterjee and Eyigungor, 2012), combined with debt renegotiation (Mihalache, 2020), and both transitory and trend endowment shocks as in Aguiar and Gopinath (2006). The model uses taste shocks (Mihalache, 2020; Dvorkin et al., 2021; Mihalache, 2025) to introduce

[3]MathWorks’ official GitHub page (MathWorks Reference Architectures) also offers pre-built AMIs ready for deployment on major commercial clouds.

convergence in the long-term debt framework with discrete choices. Such a model exemplifies the complexity in modern quantitative modeling, but still in small-medium size.

3.1 Value Function Formulation

We denote $\mathbf{s} = (z, \gamma)$ as a vector of cyclical and trend endowment shocks, and $\tilde{\beta} \equiv \beta\gamma^{1-\sigma}$ as the trend-adjusted discount factor. In this economy, a benevolent government decides whether to repay or default:

$$V(b, \mathbf{s}) = \max_{d \in \{0,1\}} (1-d)\, V^p(b, \mathbf{s}) + dV^d(\mathbf{s}), \tag{1}$$

where V^p is the value of repaying debts:

$$V^p(b, \mathbf{s}) = \max_{c \geq 0, b'} u(c) + \tilde{\beta}\mathbb{E}_{\mathbf{s}'|\mathbf{s}} V(b', \mathbf{s}'), \tag{2}$$

subject to the household's budget constraint:

$$c = y + q[\gamma b' - (1-\lambda)b] - [\lambda + (1-\lambda)\eta]b, \tag{3}$$

where $y = \exp(z)\gamma$ is the detrended exogenous endowment. V^d is the value of default:

$$V^d(\mathbf{s}) = u((1-\psi)y) + \tilde{\beta}\mathbb{E}_{\mathbf{s}'|\mathbf{s}}\Big[(1-\mu)V^d(\mathbf{s}') + \mu V^p(\Phi(\mathbf{s}'), \mathbf{s}')\Big], \tag{4}$$

where $1 - \mu$ is the probability that the sovereign carries bad credit record and cannot issue new debts, while with μ probability it can renegotiate a new debt owed to creditors $\Phi(s')$. ψ is the fraction of lost endowment due to default.

Regarding debt renegotiation, the sovereign's threat point is the value of autarky:

$$V^{aut}(\mathbf{s}) = u((1-\psi)y) + \tilde{\beta}\mathbb{E}_{\mathbf{s}'|\mathbf{s}} V^{aut}(\mathbf{s}'), \tag{5}$$

while creditors' highest loss is zero recovery ($\phi = 0$). Hence, the sovereign's surplus is

$$\Delta^{\mathrm{s}}(\phi, \mathbf{s}) = V^p(\phi, \mathbf{s}) - V^{aut}(\mathbf{s}), \tag{6}$$

and the creditors' surplus as

$$\Delta^c(\phi, \mathbf{s}) = \Big[\lambda + (1-\lambda)(\eta + q(\phi', \mathbf{s}))\Big]\phi, \tag{7}$$

where $\phi' = \mathcal{B}(\phi, \mathbf{s})$ and $\mathcal{B}(\phi, \mathbf{s})$ is the borrowing policy from (2). The restructured debt (ϕ) is determined by a Nash Bargaining game:

$$\Phi(\mathbf{s}) = \arg\max_{\phi} \Big[\Delta^{\mathbf{s}}(\phi, \mathbf{s})\Big]^{\alpha} \Big[\Delta^c(\phi, \mathbf{s})\Big]^{1-\alpha}, \qquad \Delta^{\mathbf{s}} \geq 0 \text{ and } \Delta^c \geq 0. \tag{8}$$

Foreign creditors take the expected default loss and recovery value into debt price:

$$q(b', \mathbf{s}) = \frac{1}{1+r^*}\mathbb{E}_{\mathbf{s}'|\mathbf{s}}\Big\{(1 - d(b', \mathbf{s}'))\Big[\lambda + (1-\lambda)(\eta + q(b'', \mathbf{s}'))\Big] + d(b', \mathbf{s}')q^d(b', \mathbf{s}')\Big\}. \tag{9}$$

where r^* is the risk-free rate, q^d is the price (recovery rate) for restructured bonds:

$$q^d(b', \mathbf{s}) = \frac{1}{1+r^*}\mathbb{E}_{\mathbf{s}'|\mathbf{s}}\Big\{(1-\mu)q^d(b', \mathbf{s}') + \mu\frac{\Delta^c(\phi, \mathbf{s}')}{b}\Big\}, \qquad \phi = \Phi(\mathbf{s}') \tag{10}$$

3.2 Numerical Implementation, Results and Discussions

To implement the VFI, we discretize the space of γ and z each into 25 equally-spaced grids, hence $N_s \equiv N_\gamma \times N_z = 625$. The probability transition matrix $\Pi(\mathbf{s}'|\mathbf{s})$ approximating $\mathbb{E}_{\mathbf{s}'|\mathbf{s}}$ is generated with the Tauchen (1986) method, as common in this strand of literature. Outstanding debts b, debt policy b' and restructured debts ϕ share a equal-space discretization within $[0.1, 1.4]$. The number of debt grids (N_b) takes 200 and 400. Calibration details are in Online Appendix C.

In Table 1, we compare the seconds spent to complete VFI across different implementations. "MATLAB parfor" and "MATLAB serial" refer to executing the VFI in loop form using native MATLAB code (scripted). The only difference between them is the use of the `parfor` instruction. MATLAB parfor is significantly slower than MATLAB serial by more than an order of magnitude, exemplifying common cases where parallelization inefficiency stems from the heavy interpretation overhead shared by threads/processors.

In contrast, MEX significantly removes the interpretation bottleneck through compilation. MEX-parfor offers speedups of over two orders of magnitude compared to MATLAB parfor,

Table 1: Runtime and relative performance for the benchmark sovereign default model

	Grid sizes for bond holdings			
	$N_b = 200$		$N_b = 400$	
Methods	Time (s)	Rel. Speed	Time (s)	Rel. Speed
1. Personal computer				
MATLAB parfor	3077	1	15988	1
MATLAB serial	101.6	30.3	372.5	42.9
MEX serial	53.1	58.0	185.9	86.0
MEX-parfor	12.0	256.6	49.7	321.8
MEX-CUDA	4.7	658.2	13.6	1179
1.1. Variable size inputs				
MEX-parfor	19.2	160	66.1	242
MEX-CUDA	5.46	563	14.3	1114
2. Cloud resource (AWS)				
MEX-parfor	10.0	308.1	32.6	490.4
MEX-CUDA	5.2	588.5	15.0	1066

Notes: $N_z = N_\gamma = 25$ in all cases. "Rel. Speed" denotes the relative speedup compared to MATLAB parfor for each value of N_b. "Variable size" allows for flexible input dimensions during code generation, with bounds: $N_b \in [1, 1000]$, $N_z \times N_\gamma \in [1, 1000]$. MEX on Cloud resource (AWS) uses fixed-size input variables. The setup of AWS instance is reported in Online Appendix B. GPU computations are conducted in double precision (FP64).

and expands this advantage as the grid size (N_b) increases. We also test MEX serial, which is the compiled version of MATLAB serial and does not explicitly parallelize on CPU. As Table 1 shows, MEX serial improves speed by roughly $2\times$ over MATLAB serial, a magnitude similar to Aruoba and Fernández-Villaverde (2018). MEX-parfor achieves an additional $4\times$ speedup over MEX serial, indicating the multithreading potential of the 8-core CPU are well-exploited.

On the other hand, since GPUs are optimized for massive parallelism with thousands of lightweight threads and high memory bandwidth, MEX-CUDA has the potential to significantly outperform CPU-based approaches. As shown in Table 1, our MEX-CUDA implementation achieves up to $1{,}000\times$ speedups over MATLAB parfor at $N_b = 400$, and more than three-fold over MEX-parfor. It is worth noting our MEX-CUDA is based on code involving nested loops and extensive conditional logic, with minimal code restructuring to use `coder.gpu.kernelfun` pragmas for efficient GPU kernel generation. We discuss intuitions for efficient use of GPU device in Section 5.

Execution times discussed above are based on fixed-size input arrays. On our PC, code generation (along with reports) for MEX-parfor takes 3 seconds, while MEX-CUDA takes 8 seconds. To avoid repeated code generation when varying grid sizes, we can enable flexible input dimensions using the `coder.varsize` pragma. While this introduces a moderate performance loss in MEX-parfor (Section 1.1 of Table 1), the loss is negligible in MEX-CUDA.

Section 2 of Table 1 reports the application of MEX on AWS. MEX-parfor runs on a state-of-the-art CPU with 16 physical cores and provides further 20%-50% speedup. MEX-CUDA on AWS provides similar speed as it on our PC, although the AWS GPU has much large memory size and higher floating-point operations per second (FLOPS) on double precision. This "weaker" performance gain is largely attributable to the complex control flow of the algorithm and the older GPU architecture on AWS, which limit effective exploitation of available FLOPS.

3.3 Selected alternative algorithms

The VFI in the previous section evaluates debt policy over a vector of discretized debt choices, a procedure henceforth referred to as the "brute-force" method. This approach is robust and straightforward to implement. In this subsection, we investigate whether the strong performance of the MEX implementation is preserved when alternative methods for solving the VFI are employed. For clarity of exposition, we adopt a one-period bond structure (See Online Appendix C) following Aguiar and Gopinath (2006) and Arellano (2008), and abstract from debt renegotiation, while maintaining the same state space (z, γ, b).

3.3.1 MEX with linear interpolation on policy

The first alternative method solves for the optimal debt issuance using linear interpolation. Specifically, it evaluates both $q(b', \mathbf{s})$ and $\mathbb{E}[V(b', \mathbf{s}')]$ along the debt choice dimension but off the grids.[4] Both the interpolation (implemented via a custom one-dimensional routine that is considerably faster than MATLAB's built-in `interp1`) and the root-finding algorithm (Brent's method) rely on deeply nested loops and conditional branching.

[4]Hatchondo et al. (2010) show that, in sovereign default models with one-period bonds, interpolation methods can achieve high accuracy with substantially fewer debt grid points.

Table 2: Runtime and relative performance for linear interpolation

Methods	Time (s)	Rel. Speed
1. Personal computer		
MATLAB parfor	36.0	1
MATLAB serial	129.7	0.28
MEX serial	41.5	0.87
MEX-parfor	313.6	0.11
MEX-CUDA	2.92	12.3
2. Cloud resource (AWS)		
MATLAB parfor	21.7	1.66
MEX-parfor	33.4	1.08
MEX-CUDA	2.70	13.3

Notes: In all cases, the number of grid points for state variables is fixed ($N_z = N_\gamma = 25$, $N_b = 80$). "Rel. Speed" denotes relative speeds with respect to MATLAB parfor. GPU computations use double precision. The cloud (AWS) configuration is reported in Online Appendix B.

Table 2 reports the results. MEX-CUDA remains highly effective for interpolation, delivering roughly a 12-fold speedup relative to MATLAB parfor. Interestingly, although compilation provides some benefit (MEX serial runs over three times faster than MATLAB serial), MEX-parfor performs worse than MATLAB parfor. Compiled interpolation and root-finding routines exceed the available per-core stack memory, resulting in slower memory fetches and reduced execution speed. This reflects the ineffectiveness of the compiler in optimizing execution in this specific scenario.

In contrast, MATLAB parfor interprets these functions dynamically at runtime, allowing for more efficient memory management in this special case.[5] On a higher-performance CPU (AWS setup), MEX-parfor's speed improves substantially but still falls behind MATLAB parfor.

This large memory overhead also affects MEX-CUDA. MATLAB's GPU Coder reports that the interpolation and root-finding routines occupy "too much stack memory", which can significantly constrain GPU performance. Nevertheless, MEX-CUDA still delivered substantial speedups over native MATLAB and CPU-based MEX. This reflects both the adaptability

[5]When the number of grid points for either exogenous shocks or the endogenous state variable is reduced, the relative performance of MEX-parfor improves and can exceed that of MATLAB parfor. We also experimented with compiling MEX-parfor using GCC (C/C++), which was even slower than the benchmark compiler, Visual Studio 2019.

of the CUDA compiler and the robustness of MATLAB's compilation pipeline. Performance improves further with more powerful GPU hardware on AWS.

3.3.2 MEX with binary monotonicity method

In this subsection, we evaluate MEX using the binary monotonicity method proposed by Gordon and Qiu (2018). This method exploits the monotonicity of the optimal debt choice with respect to debt states, thereby reducing the number of objective function evaluations required to solve the model. Given N_b debt states and N_b choices, this approach reduces the total evaluations from N_b^2 (as in a brute-force method) to at most $N_b \log_2(N_b) + 5N_b$. [6]

Table 3 reports main results. First, taking brute force (Panel 1) alone, the performance gains from MEX implementations are less pronounced in this simplified one-period debt model than in the more complex benchmark model (Section 1). As computational complexity decreases, MATLAB parfor becomes more efficient and tends to outperform both MATLAB serial and MEX serial.

Second, comparing Panel 1 and Panel 2, the binary monotonicity method substantially accelerates computation across all implementations, though the relative gain is smaller for MEX-parfor. As in the interpolation case, MEX-parfor exhibits high stack-memory usage due to nested branching and function calls, whereas MATLAB parfor benefits from lighter memory overhead through dynamic runtime interpretation. Switching to a higher-cache CPU on AWS does not materially improve the relative performance of MEX-parfor.

Third, MEX-CUDA consistently outperforms both CPU-based MATLAB and MEX implementations. While the advanced GPU on AWS performs markedly better than the PC GPU for brute force, its edge is less distinct in the binary monotonicity case.[7] Overall, MEX-parfor performs well when algorithmic overhead is moderate, while MEX-CUDA remains efficient across a wider range of algorithms.

[6]Although our implementation is based on a one-period debt model, similar performance is expected to hold in long-term debt models (Gordon, 2019).

[7]Several reasons can contribute to this difference. First, the binary monotonicity algorithm in Gordon and Qiu (2018) is highly serial and inherently difficult to parallelize. Second, the high stack-memory usage in the binary monotonicity case (and in previous interpolation case) slows down memory sharing. Comparing the performance of MEX-parfor and MEX-CUDA, the GPU compiler obviously does a much better job in handling this issue. Third, there are architectural differences between GPU devices. Our PC GPU (NVIDIA RTX 3060) is one generation newer than the AWS GPU (NVIDIA A10G), and improvements in stack-memory access may yield differing levels of computational efficiency.

Table 3: Runtime and relative performance for solving one-period defaultable debt model with brute force and binary monotonicity method

	Grid sizes for bond holdings			
	$N_b = 200$		$N_b = 400$	
Methods	Time (s)	Rel. Speed	Time (s)	Rel. Speed
1. **Brute force**				
1.1. *Personal computer*				
MATLAB parfor	34.1	1	99.7	1
MATLAB serial	119.0	0.29	433.0	0.23
MEX serial	55.6	0.61	229.2	0.44
MEX-parfor	26.1	1.31	84.3	1.18
MEX-CUDA	9.63	3.54	40.7	2.45
1.2. *Cloud resource (AWS)*				
MATLAB parfor	24.1	1.41	65.6	1.52
MEX-parfor	25.1	1.34	81.5	1.22
MEX-CUDA	5.46	6.24	20.7	4.82
2. **Binary monotonicity**				
2.1. *Personal computer*				
MATLAB parfor	11.6	2.93	15.9	6.27
MATLAB serial	5.6	6.07	10.8	9.21
MEX serial	24.3	1.40	46.9	2.13
MEX-parfor	23.1	1.47	44.5	2.24
MEX-CUDA	2.81	12.1	5.48	18.2
2.2. *Cloud resource (AWS)*				
MATLAB parfor	15.9	2.18	19.5	5.12
MEX-parfor	23.0	1.48	44.2	2.26
MEX-CUDA	2.87	11.9	5.40	18.5

Notes: The number of grid points for exogenous endowment shocks is fixed at $N_z = N_\gamma = 25$ in all cases. "Rel. Speed" denotes relative speed compared to MATLAB parfor with brute-force search. All GPU computations use double precision. The cloud (AWS) configuration is reported in Online Appendix B.

3.3.3 Vectorization

Native MATLAB is widely regarded efficient when operations are vectorized, while less efficient when executing nested loops. Although complex structures in economic models are more naturally coded using loops (like our benchmark sovereign default model), it is nevertheless informative to benchmark the performance of vectorized computation.

In this experiment, 'MATLAB vec' denotes vectorized computation on the CPU. 'gpuArray' refers to MATLAB's native GPU implementation of vectorized operations that does not involve code generation; it performs operations on GPU by transferring matrices to GPU memory using the `gpuArray()` command. 'MEX-CUDA vec' represents the compiled version of the gpuArray approach. These vectorized methods are equivalent in logic to the brute-force in Section 3.3.2.

Table 4: Runtime for solving one-period debt model with fully vectorized VFI.

	Grid sizes for bond holdings			
	$N_b = 200$		$N_b = 400$	
Methods	Time (s)	Rel. Speed	Time (s)	Rel. Speed
1. Personal computer				
MATLAB vec	74.0	1	287.0	1
gpuArray	12.3	6.00	427.9	0.67
MEX-CUDA vec	9.68	7.65	43.1	6.67
MEX-CUDA	9.63	7.69	40.7	7.04
2. Cloud resource (AWS)				
gpuArray	13.1	5.65	47.4	6.06
MEX-CUDA vec	5.31	12.7	20.7	13.9
MEX-CUDA	5.46	13.6	20.7	13.9

Notes: $N_z = N_\gamma = 25$ in all cases. MEX-CUDA vec refers to CUDA-generated code based on the vectorized version of the algorithm. MEX-CUDA results are copied from the brute-force case reported in Table 3. "Rel. Speed" denotes performance relative to MATLAB vec. All GPU computations use double precision. The cloud (AWS) configuration is reported in Online Appendix B.

As Table 4 reports, MATLAB vec is outperformed by both MATLAB and MEX implementations using `parfor` (Panel 1 of Table 3). Employing gpuArray substantially improves performance at $N_b = 200$ but leads to a sharp slowdown at $N_b = 400$ on our PC. Compiling the vectorized code (MEX-CUDA vec) delivers roughly a tenfold speedup over gpuArray at $N_b = 400$, and performs similar to MEX-CUDA on both our PC and AWS.[8] Overall, vectorized implementations in scripted code fail to outperform their compiled, loop-based counterparts.

From a computational perspective, vectorization requires extensive matrix reshaping and

[8]For the vectorized version, CPU-based MEX compilation does not yield meaningful performance gains and is therefore omitted from Table 4.

temporary memory allocations, which impose substantial memory overhead. In contrast, compiler-optimized loop structures exploit more efficient memory-access patterns, mitigating the bottlenecks that often arise in large-scale economic models. Moreover, the vectorized code consumes substantially more memory: when $N_b > 500$, our PC GPU's 6 GB memory capacity is exhausted, preventing the computation to proceed. In contrast, loop-based implementations compiled with MEX-CUDA allocate memory dynamically and can handle grids as large as $N_b = 10{,}000$, making them far more suitable for high-dimensional models.

3.3.4 Execution Speed Comparison with Other Languages

We benchmark the execution speed of the MEX implementation against other popular languages that leverage GPU resources, namely CUDA Fortran and PyTorch. CUDA Fortran is an open-source extension of Fortran that enables computing on NVIDIA GPUs using standard Fortran syntax. Similar to MEX-CUDA, developers can offload computation of loops to the GPU by annotating existing code with directives (pragmas), rather than writing explicit kernels as in CUDA and Julia (Deng et al., 2022). PyTorch is a Python-based framework that has become one of the most widely used GPU computing libraries for deep learning applications.

Table 5 presents the runtime comparisons. Panel 1 reports results from the one-period sovereign default model solved with brute-force search. Fortran OpenMP executes parallel computation across CPU cores using the OpenMP API. CUDA Fortran uses the OpenACC API, which compiles annotated for-loops into CUDA kernels and executes on the GPU.[9] Overall, MEX and Fortran deliver comparable computational efficiency.

Panel 2 reports results for a vectorized version from Duarte et al. (2020), ensuring comparability.[10] Here, PyTorch CPU refers to computation on the CPU, while PyTorch CUDA denotes GPU execution, with VFI routines compiled via PyTorch's JIT compiler. The performance of MEX-CUDA and PyTorch CUDA is comparable, whereas MATLAB executes

[9]As of the latest release, CUDA Fortran does not support the Windows operating system; we therefore conduct its benchmark on Ubuntu 24.04 LTS on the same PC. All other experiments are conducted on Windows 11 OS.

[10]Unlike Duarte et al. (2020), who benchmark PyTorch and TensorFlow using single precision (FP32) but employ double precision for other languages, we use double precision (FP64) consistently across all implementations. While GPU execution in FP32 can be over twenty times faster than in FP64 on our hardware, FP64 is preferred in quantitative macroeconomic modeling for numerical accuracy. For example, in solving our benchmark sovereign default model with long-maturity bonds, FP32 often leads to failed convergence or results that deviate noticeably from FP64.

Table 5: Runtime (in seconds) comparison across selected languages

1. One-period debt model	$N_b = 200$	$N_b = 400$
MEX-parfor	25.7	61.3
MEX-CUDA	9.63	40.7
Fortran OpenMP	18.8	59.4
CUDA Fortran	11.0	43.2
2. Duarte et al. (2020) model	$N_b = 500$	$N_b = 1000$
MATLAB vec	101.4	403.3
PyTorch CPU	191.7	861.5
MEX-CUDA	31.9	124.7
PyTorch CUDA	30.6	119.4

Notes: We solve sovereign default models with one-period bonds using MATLAB MEX, Fortran, and PyTorch implementations. For the comparison with Fortran (Panel 1), we employ the same benchmark model as in Table 3 using brute-force, loop-based code with $N_z = N_\gamma = 25$. MATLAB vec refers to scripted, CPU-based MATLAB code. For the PyTorch comparison (Panel 2), we adopt the sovereign default model from Duarte et al. (2020), where the main calculations are vectorized.

significantly faster on CPU as reported in Aruoba and Fernández-Villaverde (2015). Moreover, the vectorized implementation itself is markedly slower than the loop-based version (the upper panel of Table 5) of the same model.

4 Experiment 2: Heterogeneous agents model

This section presents computational experiments based on the heterogeneous-agent framework of Buera and Shin (2013). Their model extends an otherwise standard neoclassical growth model by incorporating entrepreneurship and financial frictions. Individuals differ in their asset holdings and entrepreneurial productivity, and in each period choose whether to operate a business or work as a salaried employee. This framework has been widely used in the macroeconomic literature with heterogeneous agents and financial frictions (see, for example, Buera et al. (2015) for a review of development applications and Buera et al. (2021) and Morazzoni and Sy (2022) for recent contributions).

From a technical point of view, the occupational choice makes the agents' value function not concave: a kink in the value function arises at the point of indifference between the two occupational choices. This creates significant hurdles for researchers attempting to use

Euler-equation-based methods.[11]

4.1 The model

Time is discrete and the horizon is infinite. Individuals are heterogeneous in their wealth a and in their entrepreneurial productivity z. Entrepreneurial productivity z evolves stochastically according to a first-order Markov process with transition matrix $Q(z, z')$. The individual's problem can be written recursively as

$$V(a, z) = \max_{c,a'} \left\{ \frac{c^{1-\gamma}}{1-\gamma} + \beta \mathbb{E}\Big[V(a', z') \mid z \Big] \right\}, \tag{11}$$

subject to

$$c + a' \leq \max \Big\{ w, \pi(a, z|r, w) \Big\} + (1 + r)a, \quad a' \geq 0, \tag{12}$$

where r and w are the interest rate and wage, respectively, and $\pi(a, z \mid r, w)$ denotes entrepreneurial profits:

$$\pi(a, z|r, w) = \max_{l,k} \{ f(z, k, l) - wl - (r + \delta)k \}, \quad \text{s.t.} \quad k \leq \lambda a, \tag{13}$$

with production function

$$f(z, k, l) = z(k^{\alpha} l^{1-\alpha})^{1-\nu}, \tag{14}$$

where $\nu \in (0, 1)$ is the span-of-control parameter. The solution yields the savings policy $a'(a, z)$, the optimal factor demand policies $k(a, z)$, $l(k, z)$ and the occupational choice $e(a, z) \in \{0, 1\}$:

$$e(a, z) = \begin{cases} 1, & \text{if } \pi(a, z \mid r, w) > w, \quad \text{(entrepreneur)} \\ 0, & \text{otherwise.} \quad \text{(worker)} \end{cases} \tag{15}$$

We compute the stationary equilibrium using value function iteration (VFI) nested inside an outer fixed-point routine over prices (r, w):

1. **Discretize the state space.** Construct grids for wealth $a \in \{a_1, \ldots, a_{N_a}\}$ and en-

[11] Fella (2014) shows how to generalize the endogenous gridpoint method (EGM) to non-concave problems. As Gortz and Mirza (2018) point out, however, it is quite complex to implement the generalized EGM, compared to the straightforward VFI method.

trepreneurial ability $z \in \{z_1, \ldots, z_{N_z}\}$. See Appendix D.1.2 for further details on the discretization of z.

2. **Value function iteration (inner loop).** For given prices (r, w), we solve for $a'(a, z)$ and $e(a, z)$ using VFI with Howard improvement. Before running the VFI, we analytically solve the entrepreneurial profit maximization problem:

$$\pi(a, z \mid r, w) = \max_{k,l}\{f(z, k, l) - wl - (r + \delta)k\}, \qquad k \leq \lambda a,$$

which yields the optimal factor demands $k(a, z)$ and $l(a, z)$. We then use this solution to precompute the static return function $R(a', a, z)$ for the Bellman update. For iteration $n = 0, 1, \ldots,$ the Bellman equation is

$$V_{n+1}(a, z) = \max_{a'}\left\{R(a', a, z) + \sum_{z'} V_n(a', z')\, Q(z, z')\right\},$$

where

$$R(a', a, z) = \begin{cases} \dfrac{(c(a', a, z))^{1-\gamma}}{1-\gamma} & \text{if } c > 0, \\ -\infty & \text{if } c \leq 0, \end{cases}$$

and

$$c(a', a, z) = \max\{w,\ \pi(a, z \mid r, w)\} + (1 + r)a - a'.$$

We implement modified policy iteration (Howard improvement) to accelerate convergence of the VFI.

3. **Stationary distribution.** Given the policy function $a'(a, z)$ and the exogenous transition $Q(z, z')$, obtain the stationary distribution $\mu(a, z)$.

4. **Aggregation and market clearing.** Using $\mu(a, z)$ and $e(a, z)$, compute aggregates and excess-demand residuals:

$$ED^K(r, w) = \int k(a, z)\mathbb{I}_{\{e(a,z)=1\}}\, d\mu(a, z) - \int a\, d\mu(a, z), \tag{16}$$

$$ED^L(r, w) = \int l(a, z)\mathbb{I}_{\{e(a,z)=1\}}\, d\mu(a, z) - \int \mathbb{I}_{\{e(a,z)=0\}}\, d\mu(a, z), \tag{17}$$

where (16) denotes excess demand for capital and (17) excess demand for labor.

5. **General equilibrium (outer loop).** Update (r, w) until both $|ED^K|$ and $|ED^L|$ are below tolerance. In practice, we minimize the squared residuals using MATLAB's `fminsearch` routine:

$$(r^*, w^*) = \arg\min_{r,w} \left\{ (ED^K(r,w))^2 + (ED^L(r,w))^2 \right\}. \tag{18}$$

Further details on analytical derivations, definition of equilibrium and calibration are provided in Online Appendix D.

4.2 Computation implementation

Table 6: Runtime and relative performance for solving the heterogeneous agents model

	Grid sizes for asset holdings			
	$N_a = 1001$		$N_a = 1501$	
Methods	Time (s)	Rel. Speed	Time (s)	Rel. Speed
1. Personal computer				
MATLAB parfor	1128.6	1	6555.9	1
MATLAB serial	190.7	5.92	265.2	24.7
MEX serial	136.2	8.29	238.7	27.5
MEX-parfor	29.1	38.8	48.4	135.5
MEX-CUDA	15.6	72.5	22.7	288.6
1.1. Variable-size inputs				
MEX-parfor	42.7	26.4	102.8	63.8
MEX-CUDA	16.0	70.7	23.2	282.5
2. Cloud resource (AWS)				
MEX-parfor	24.5	46.1	38.1	172.3
MEX-CUDA	11.2	101.1	17.6	371.5

Notes: $N_z = 40$ in all cases. "Rel. Speed" denotes relative speeds to the MATLAB parfor for a given N_a. GPU computations use double precision. "Variable size" indicates that, at code generation, input sizes to the solver are not fixed but bounded by $N_a \in [1, 2001]$. The cloud (AWS) configuration is reported in Online Appendix B.

We parallelize only the most computationally intensive components of the algorithm—namely, the calculation of $R(a', a, z)$, the VFI, and the stationary distribution $\mu(a, z)$. Aggregation and market-clearing operations within the inner loop, as well as the outer-loop general equilibrium search, are executed on the CPU using standard MATLAB scripts without explicit parallelization directives (MATLAB serial). This hybrid implementation delivers the best overall performance.

Table 6 reports runtime comparisons for the heterogeneous-agent model. MATLAB parfor performs poorly for constructing and using $R(a', a, z)$ and therefore lags behind both MATLAB/MEX serial implementations. Using the same parallelization paradigm as MATLAB parfor, MEX-parfor and MEX-CUDA accelerate execution by between 39× and 289×, with gains increasing in N_a. When allowing variable-size inputs, MEX-parfor slows markedly, whereas MEX-CUDA remains essentially unchanged relative to its fixed-size version. Faster CPU and GPU hardware on AWS further improves MEX performance.

We also benchmark fully vectorized computation in native MATLAB (Table 7). "MATLAB vec" denotes CPU-based scripted code execution, and "gpuArray" denotes MATLAB's native GPU execution. These methods outperform native loop-based MATLAB (Table 6) but remain slower than MEX parallelization on both the PC and AWS.

Table 7: Runtime for solving heterogeneous agents model with fully vectorized VFI.

	Grid sizes for asset holdings					
	$N_a = 1001$		$N_a = 1501$		$N_a = 2001$	
Methods	Time (s)	Rel. Speed	Time (s)	Rel. Speed	Time (s)	Rel. Speed
1. Personal computer						
MATLAB vec	90.8	1	155.1	1	262.7	1
gpuArray	23.7	3.82	26.9	5.77	98.2	2.67
MEX-parfor	29.1	3.12	48.4	3.21	103.7	2.53
MEX-CUDA	15.6	5.83	22.7	6.83	38.3	6.85
2. Cloud resource (AWS)						
MATLAB vec	74.9	1.12	105.8	1.47	163.3	1.61
gpuArray	21.8	3.43	23.8	4.45	28.6	9.18
MEX-parfor	24.5	3.71	38.1	4.08	67.4	3.90
MEX-CUDA	11.2	8.14	17.6	8.79	28.3	9.27

Notes: $N_z = 40$ in all cases. "Rel. Speed" denotes performance relative to MATLAB vec. MEX-CUDA results are reproduced from Table 6. Because of memory requirements, MEX-CUDA vec (compiled vectorized code) runs out of memory for $N_a \geq 1501$ and is omitted. GPU computations use double precision. The cloud (AWS) configuration is reported in Online Appendix B.

On AWS, the relative performance of gpuArray compared to MEX-CUDA improves as the asset-grid size N_a increases, whereas no clear pattern on the PC. The larger GPU memory on AWS (24 GB versus 6 GB on our PC) alleviates the memory bottlenecks that typically slow down vectorized operations when array sizes approach the limits of available memory. As N_a grows further, MEX-CUDA once again becomes substantially faster than gpuArray on the AWS GPU.

5 Kernel creation and memory management in efficient GPU computation

While MEX-CUDA can deliver substantial execution speed, it requires additional development efforts. That said, the effort is still far lower than writing explicit CUDA kernels (or reimplementing GPU code in other languages), because MEX-CUDA does not require hand-written kernels. Instead, developers annotate MATLAB code with compiler directives (pragmas) that guide memory placement and thread-level parallelism, thereby exploiting the advantages of GPU computing.[12] This section presents representative code blocks from the benchmark sovereign default model to illustrate kernel generation via pragmas and loop design.[13]

Listing 2 shows the computation of V^p (see Equation (2)), the repayment value function. Several adjustments make the MATLAB code more suitable for MEX-CUDA. First, `coder.gpu.kernelfun` maps the function to a GPU kernel entry point. Second, we ensure all operations are element-wise and place `coder.gpu.kernel()` immediately before the outer loop to ensure the nested loops are mapped into GPU kernels. Third, the expectation term $\mathbb{E}_{\mathbf{s}'|\mathbf{s}}V$ is written as matrix arithmetic, which the compiler maps to efficient CUDA kernels. Fourth, `coder.nullcopy` prevents unnecessary memory copying during array initialization, and `coder.gpu.constantMemory` places read-only inputs in constant memory, enabling fast broadcast to all GPU threads.

Listing 3 implements the price mapping with taste shocks (see Equation (9)). Rather than materializing a temporary vector of exponentials $\{\texttt{theExp}\}_{i=1}^{N_b}$, the code accumulates `sumExp`

[12]An illustration of why GPUs handle massive parallelism more efficiently than CPUs is provided in Appendix E.

[13]Pragmas are compiler directives that influence how code is lowered during compilation. They are ignored in native MATLAB execution. See MATLAB's documentation on Kernel Creation from MATLAB Code.

```
function [vp,vd,q,bp,bpr,pdef,def,rr] = solver_gpu(z,m,b,pdf,para)
coder.gpu.kernelfun
W = coder.nullcopy(zeros(ns,nb,nb)); vp = coder.nullcopy(zeros(ns,nb));
while diff > tol && its< maxits
evp = m.^(1-sigg).*betta.*pdf*V ;
coder.gpu.kernel()
for is = 1 : ns  % to calculate the value of default
    coder.gpu.constantMemory(z) ; coder.gpu.constantMemory(m) ;
    y = exp(z(is))*m(is) ;
    for ib = 1 : nb
        coder.gpu.constantMemory(b) ;
        for i = 1 : nb
            if q(is,i) >= 0.45
            c = y - (eta+(1-eta)*coup)*b(ib) + q(is,i)*(b(i)*m(is)...
            - (1-eta)*b(ib)) ;
              if c <= 0;  W(i, ib, is) = - Inf ;
              else
              W(i,ib,is) = (c.^(1-sigg)-1)/(1-sigg) + evp(is,i);
              end
            else; W(i, ib, is) = - Inf ;
            end
        end
        [vpnew(is,ib), bp(is,ib)] = max( W(:, ib, is) ) ;
    end
end
end
```

Listing 2: MATLAB code to calculate the value of repayment V^p in the sovereign default model, which is part of the VFI solver. We restrict the minimum debt price to 0.45 to eliminate consumption peaks prior to default.

```
coder.gpu.kernel()
for is = 1:ns % to incorporate taste shocks on debt choice
    for ib = 1:nb
        sumExp = 0;  sumExpQ = 0;
        for i = 1:nb
            temp = W(i, ib, is) - vpnew(is, ib) - cv_bp;
            if temp > 0
                theExp = exp((temp + cv_bp) / sigg_bp);
                sumExp = sumExp + theExp;
                sumExpQ = sumExpQ + theExp * q(is, i);
            end
        end
        qnew(is, ib) = eta + (1 - eta) * (coup + sumExpQ / sumExp);
    end
end
```

Listing 3: MATLAB code to incorporate taste shocks on discrete debt choices, which facilitates the convergence of VFI algorithm.

and `sumExpQ` on the fly. This design minimizes per-thread stack usage within the kernel by avoiding large temporaries. On GPUs, each thread has limited fast storage (registers and a small stack/local space), and overflowing these resources spills to slower heap memory. Excessive stack pressure can therefore reduce occupancy and throttle throughput. By contrast, CPUs run fewer threads with larger stacks, so the same pattern is less problematic on the CPU.

In Listings 2 and 3, we also separate the formation and consumption of the welfare array `W(i,ib,is)` into distinct kernels. This lowers per-kernel stack usage at the cost of allocating the large array twice – a trade-off that proved favorable in practice.

Overall, the required "GPU expertise" for MEX-CUDA remains accessible: add the key pragmas, ensure loop bodies are element-wise and branch-light where possible, and refactor to avoid memory-heavy temporaries. MATLAB Coder's code generation reports also assist with identifying potential performance bottlenecks (See Line 3 in Listing 1).

6 Conclusion

Using the MATLAB Executable (MEX) framework together with a range of algorithms, this study has explored CPU and GPU parallelization to accelerate the solution of complex nonlinear macroeconomic models. Our results show that the MEX approach can deliver speed improvements of up to 1,000-fold over native MATLAB execution, reaching performance comparable to established alternatives such as CUDA Fortran and PyTorch. Importantly, this approach does not require prior experience with Fortran, CUDA, or C/C++: users simply add the appropriate pragmas or directives to functions written in native MATLAB and generate C/CUDA code that executes seamlessly within the MATLAB environment.

Moreover, MEX methods outperform vectorized native MATLAB tools, such as `gpuArray`, because they optimize loop execution and memory management by leveraging mature third-party compilers, most notably the CUDA Toolkit and Microsoft Visual Studio, during code generation. This loop-based design not only enhances speed but also reduces memory demands, enabling the inclusion of richer and more complex model features. We also demonstrate that MEX implementations can be readily deployed on cloud platforms such as Amazon Web Services, allowing researchers to solve large-scale nonlinear economic models efficiently using advanced hardware.

In summary, MEX provides an accessible and powerful framework for harnessing the parallel computing capabilities of modern hardware. It enables a wide range of economic researchers to achieve substantial computational gains without departing from scripted programming practices.

Acknowledgements

We thank Christoph Görtz, Oliver De Groot, Juan Francisco Rubio-Ramirez, Robert Kirkby for their comments. Declarations of interest: none. All errors are our own.

Declaration

Conflict of interest: none.

References

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., 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., 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.

Aruoba, S. B. and Fernández-Villaverde, J. (2018). A comparison of programming languages in macroeconomics: an update. Technical report, University of Pennsylvania.

Buera, F. J., Fattal-Jaef, R. N., Hopenhayn, H., Neumeyer, P. A., and Shin, Y. (2021). The economic ripple effects of covid-19. NBER Working Papers 28704, National Bureau of Economic Research, Inc.

Buera, F. J., Kaboski, J. P., and Shin, Y. (2015). Entrepreneurship and financial frictions: A macrodevelopment perspective. *Annual Review of Economics*, 7(1):409–436.

Buera, F. J. and Shin, Y. (2013). Financial frictions and the persistence of history: A quantitative exploration. *Journal of Political Economy*, 121(2):221–272.

Carroll, C. D. (2006). The method of endogenous gridpoints for solving dynamic stochastic optimization problems. *Economics letters*, 91(3):312–320.

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.

Duarte, V., Duarte, D., Fonseca, J., and Montecinos, A. (2020). Benchmarking machine-learning software and hardware for quantitative economics. *Journal of Economic Dynamics and Control*, 111:103796.

Dvorkin, M., Sánchez, J. M., Sapriza, H., and Yurdagul, E. (2021). Sovereign debt restructurings. *American Economic Journal: Macroeconomics*, 13(2):26–77.

Fella, G. (2014). A generalized endogenous grid method for non-smooth and non-concave problems. *Review of Economic Dynamics*, 17(2):329–344.

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

Gordon, G. (2019). Efficient computation with taste shocks. Technical report, Federal Reserve Bank of Richmond.

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

Gortz, C. and Mirza, A. (2018). Solving models with jump discontinuities in policy functions. *Oxford Bulletin of Economics and Statistics*, 80(2):434–456.

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.

Kirkby, R. (2017). A toolkit for value function iteration. *Computational Economics*, 49(1):1–15.

Mendoza, E. G. and Villalvazo, S. (2020). Fipit: A simple, fast global method for solving models with two endogenous states & occasionally binding constraints. *Review of Economic Dynamics*, 37:81–102.

Mihalache, G. (2020). Sovereign default resolution through maturity extension. *Journal of International Economics*, 125:103326.

Mihalache, G. (2025). Solving default models. *Oxford Research Encyclopedia of Economics and Finance*, 0.

Morazzoni, M. and Sy, A. (2022). Female entrepreneurship, financial frictions and capital misallocation in the us. *Journal of Monetary Economics*, 129:93–118.

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.

Peri, A. (2020). A hardware approach to value function iteration. *Journal of Economic Dynamics and Control*, 114:103894.

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

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

Online Appendix

Efficient Parallelization of Nonlinear Macroeconomic Models: A MATLAB Executable Approach

by Alessandro Di Nola, Liang Shi

January 2025

A Supplementary data

Supplementary MATLAB code can be found in authors' GitHub repositories.

B Computation Platform

Our benchmark results are obtained on our personal computer. Relevant hardware and software are as follows:

1. Intel Core i7-11800H CPU with 8 physical cores;
2. Dynamic Random Access Memory (DRAM) of 16 GB operating at 3200 MHz;
3. NVIDIA GeForce RTX 3060 GPU (Laptop version) with 6 GB GPU memory (GRAM);
4. MATLAB 2023b with Parallel Computation Toolbox, MATLAB Coder and MATLAB GPU coder;
5. Microsoft Windows 11 Pro operating system;
6. GPU driver: NVIDIA GeForce Game Ready Driver, Release 580 Driver for Windows, Version 581.29;
7. Compilers: Windows Visual Studio 2019 and CUDA Toolkit 12.0. See *Supported and Compatible Compilers for GPU Coder* website maintained by the MathWorks.

We also use Amazon Web Services (AWS) to exemplify using MEX on cloud and cluster resources, relevant hardware and software are as follows:

1. For MEX-parfor, we use AWS EC2 'C6a.8xlarge' instance utilizing 16 physical cores of an EPYC 7R13 CPU and 64 GB DRAM;
2. For MEX-CUDA, we use an AWS EC2 'G5.xlarge' instance utilizing a NVIDIA A10G Tensor Core GPU with 24 GB GRAM and 2 CPU cores with 16 GB DRAM;
3. For both cases, the OS is Microsoft Windows Server 2022 Base;
4. All other setups are the same as our personal computer.

C Calibration of the Sovereign Default Model

In the quantitative implementation of sovereign default model, the utility function takes standard constant relative risk aversion (CRRA) form:

$$U(c) = \frac{c^{1-\sigma} - 1}{1 - \sigma} \tag{19}$$

The transitory shock z and γ follows AR(1) processes:

$$z_t = \rho_z z_{t-1} + \varepsilon_t^z, \quad \varepsilon_t^z \sim i.i.N(0, \sigma_z^2) \tag{20}$$

$$\gamma_t = (1-\rho_\gamma)\overline{\gamma} + \rho_\gamma \gamma_{t-1} + \varepsilon_t^\gamma, \quad \varepsilon_t^\gamma \sim i.i.N(0, \sigma_\gamma^2) \tag{21}$$

Parameters for taste shocks on debt policy σ^b, default σ^d and renegotiation σ^ϕ are all 5×10^{-5} so that they are kept minimum while enabling the VFI to finish within 500 iterations. The while-loop for VFI is terminated when the sum of maximum difference of V^r, V^d and q in $n-1$ and n iterations are smaller than 10^{-7}.

The sovereign default model with one-period debt discussed in Section 3.3 is a simplified version of the benchmark model. Specifically, the budget constraint for repayment is:

$$c = y + \gamma q b' - b, \tag{22}$$

which is a special case of Equation (3) for $\eta = 0$ and $\lambda = 1$. Due to the abstraction from debt restructuring ($\phi = 0$), the one-period debt model's value function of default becomes:

$$V^{d,one}(\mathbf{s}) = u((1-\psi)y) + \tilde{\beta}\mathbb{E}_{\mathbf{s}'|\mathbf{s}}\left[(1-\mu)V^{d,one}(\mathbf{s}') + \mu V^p(0, \mathbf{s}')\right] \tag{23}$$

Table C.1: Calibration of the benchmark sovereign default model

Description	Parameters	Value
Risk aversion	σ	2
Subjective discount	β	0.83
Probability of bargaining	μ	0.154
Endowment loss	ψ	10%
Coupon rate	η	0.06
Reciprocal of maturity	λ	0.15
Quarterly risk-free rate	r^*	4%
z shock AR(1)	ρ_z	0.90
SD of z shock	σ_z	3%
γ shock AR(1)	ρ_γ	0.90
SD of γ shock	σ_γ	0.75%
Average trend	$\overline{\gamma}$	1.01
Bargaining power	α	0.6
Debt issuance taste	σ^b	0.001
Default taste	σ^d	0.001
Renegotiation taste	σ^ϕ	0.001

Notes: Parameters target annual frequency.

Without the value of restructured debts, the one-period debt pricing rule is:

$$q^{one}(b', \mathbf{s}) = \frac{1}{1+r^*}\mathbb{E}_{\mathbf{s}'|\mathbf{s}}(1 - d(b', \mathbf{s}'))q^{one}(b'', \mathbf{s}') \tag{24}$$

which is a special case of Equation (9) when the price of restructured debt q^d does not exist.

D Heterogeneous Agents Model Details

D.1 Analytical Solution

The entrepreneur's profit maximization problem (see eq. 13 in the paper) can be solved analytically:

$$k^{unc}(z) = \left[\left(\frac{\alpha(1-\nu)z}{r+\delta}\right)^{1-(1-\alpha)(1-\nu)}\left(\frac{(1-\alpha)(1-\nu)z}{w}\right)^{(1-\alpha)(1-\nu)}\right]^{1/\nu}, \tag{25}$$

$$k(a, z) = \min\{k^{unc}(z), \lambda a\}. \tag{26}$$

$$l(a,z) = \left[\frac{(1-\alpha)(1-\nu)z}{w}\,(k(a,z))^{\alpha(1-\nu)}\right]^{\frac{1}{1-(1-\alpha)(1-\nu)}}. \tag{27}$$

$$y(a,z) = z(k(a,z)^{\alpha} l(a,z)^{1-\alpha})^{1-\nu} \tag{28}$$

$$\pi(a,z \mid r,w) = y(a,z) - wl(a,z) - (r+\delta)k(a,z). \tag{29}$$

D.1.1 General Equilibrium Conditions

There are two general equilibrium prices: interest rate r and wage rate w. Market clearing requires:

$$\int l(a,z)\cdot \mathbb{I}_{\{e(a,z)=1\}}\, d\mu(a,z) = \int \mathbb{I}_{\{e(a,z)=0\}}\, d\mu(a,z) \quad \text{(labor)}, \tag{30}$$

$$\int k(a,z)\, \mathbb{I}_{\{e(a,z)=1\}}\, d\mu(a,z) = \int a\, d\mu(a,z) \quad \text{(capital)}. \tag{31}$$

The market clearing condition for the consumption good is implied by Walras' law:

$$\int c(a,z) d\mu(a,z) = Y - \delta K \tag{32}$$

where

$$Y = \int y(a,z)\mathbb{I}_{\{e(a,z)=1\}} d\mu(a,z) \tag{33}$$

and

$$K = \int a\, d\mu(a,z). \tag{34}$$

D.1.2 Calibration

External parameters. $\gamma = 1.5$, $\delta = 0.06$, $\alpha = 0.33$.

Internal calibration. Choose ν (span of control), (η, ψ) (dispersion and persistence of entrepreneurial ability) and β (discount factor) to match:

- Top 10% of employment (ν, η)
- Top 5% of earnings (ν, η)
- Entrepreneurs exit rate (ψ)

- Real interest rate (β)

Stochastic process for ability. Entrepreneurial ability z follows a truncated Pareto distribution with density function

$$p(z) = \begin{cases} \eta\, z^{-(\eta+1)}, & z \geq 1, \\ 0, & \text{otherwise.} \end{cases} \tag{35}$$

Each period, with probability ψ an individual keeps z; with probability $1-\psi$ they draw $z' \sim p(\cdot)$. Thus,

$$\mathbb{E}\left[V(a', z') \mid z\right] = \psi V(a', z) + (1-\psi) \int V(a', z') p(z') dz'. \tag{36}$$

In order to re-use the general routines for value function iteration and distribution iteration, it is convenient to recast the stochastic process described above as a Markov chain. First, we assume that we have already discretized the Pareto distribution as $z \in \{z_1, z_2, \ldots, z_{N_z}\}$ and $p(z) = [p_1, \ldots, p_{N_z}]$ (a $N_z \times 1$ column vector such that $\sum_i p_i = 1$). Then, we observe that if the current ability shock is z_i, then next-period ability shock will be z_i with prob $\psi + (1-\psi)p_i$ and z_j with prob $(1-\psi)p_j$, for $j \neq i$. It is easy to check that $\sum_{z'} \pi(z_i, z') = \psi + (1-\psi)p_i + \sum_{j \neq i}(1-\psi)p_j = 1$, i.e. each row i of the Markov chain sums to one, for all $i = 1, \ldots, N_z$. Therefore the $N_z \times N_z$ Markov chain for the shock (z, z') can be written as

$$\Pi_{N_z \times N_z} = \psi \mathbf{I}_n + (1-\psi)\mathbf{1}_{N_z} p^T \tag{37}$$

or, more in detail, as

$$\Pi_{N_z \times N_z} = \psi \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 1 \end{bmatrix} + (1-\psi) \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \begin{bmatrix} p_1 & p_2 & \ldots & p_{N_z} \end{bmatrix} \tag{38}$$

where $\mathbf{I}_{N_z}$ is the $N_z \times N_z$ identity matrix and $\mathbf{1}_{N_z}$ is an $N_z \times 1$ *column* vector of ones.

For the replication we used the files `support.dat` and `dist.dat` kindly provided by the authors.

E Performance and Architecture

In this paper we observe the speedup of MEX-based parallelization varies with the number of grid points. MEX-parfor tends to perform better with smaller workloads, while MEX-CUDA is more advantageous for larger workloads. One reason for this is that CPU-based multi-threading accelerates the for-loops but not explicitly for the rest of the matrix arithmetic. 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. On the other hand, MEX-CUDA can recognize and translate vectorized matrix operations into 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 processors: In our 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, our 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 design for simple tasks, which 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.

Another significant factor is memory bandwidth and latency, as illustrated in Figure E.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.[14] 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 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

[14]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.

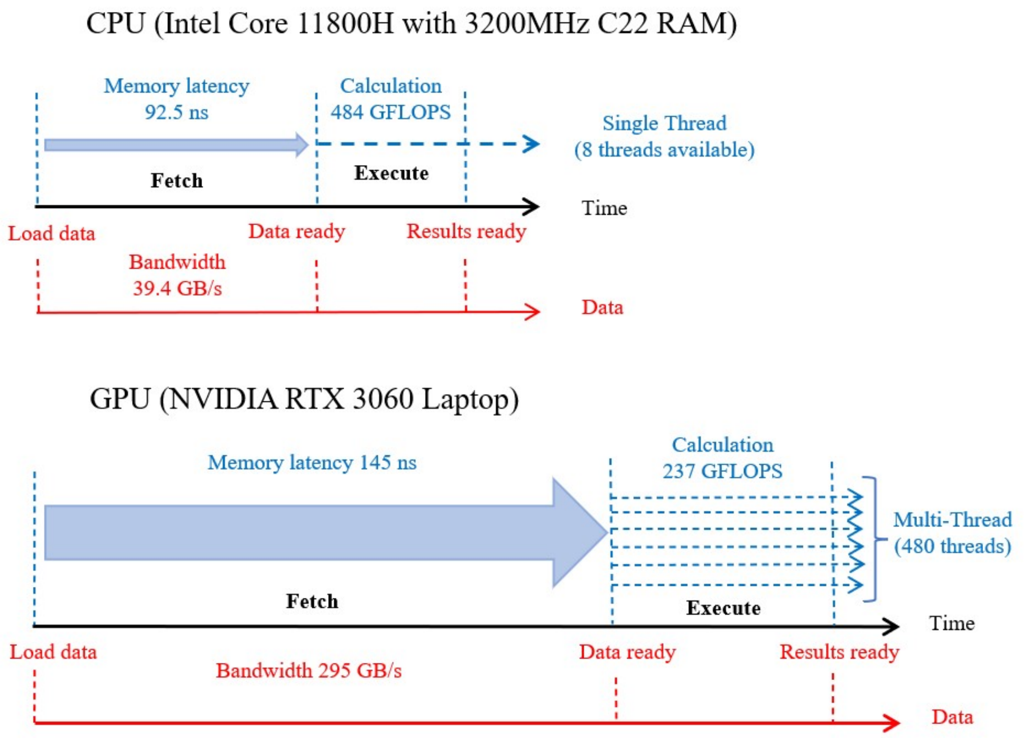

Figure E.1: An illustration of memory latency in CPU and GPU. The GFLOPSs refer to one billion floating-point operations per second in double precision. FLOPS and Memory data is tested via the AIDA64 GPGPU Benchmark and AIDA64 Cache & Memory Benchmark.

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 workloads, 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.