

Parallel Computation of Sovereign Default Models^{*}

Mingzhuo Deng[†] Pablo A. Guerron-Quintana[‡] Lewis Tseng[§]

June 15, 2022

Abstract

This paper discusses the parallel and efficient computation of macroeconomic models, with an emphasis on solving sovereign default models. Our motivation is two-fold. First, we aim to streamline complex numerical models in a parallel computation fashion. Second, we want to unleash the power of GPUs (Graphic Processing Unit) but bypass the steep learning and implementation costs of languages like C++ and CUDA (Compute Unified Device Architecture) in economic research. To this end, we propose a framework for efficient parallel computing with the modern language Julia. The paper offers a detailed analysis of parallel computing, Julia-style acceleration techniques, and coding recommendations. The Julia with CUDA benchmark shows a substantial speedup of over 1,000 times compared to standard Julia that runs on a CPU. Our Julia with CUDA's implementation is twice as fast as that of the C++ Standard Parallel Library. We provide an accompanying GitHub repository with the codes and the benchmarks used in this paper.

Keywords: Sovereign Default Model, Julia, C++, CUDA, GPU, Parallel Computation.

1 Introduction

This paper provides a framework for the efficient parallel computation of nonlinear macroeconomic models. We choose the modern programming language Julia to construct

^{*}We thank Christian Velasquez and an anonymous referee for useful comments. Pablo A. Guerron-Quintana thanks Juan Rubio-Ramirez for introducing him to the world of C/CUDA and GPUs.

[†]University of Pennsylvania, dengmz@upenn.edu

[‡]Boston College, pguerron@gmail.com. Corresponding author.

[§]Boston College, lewis.tseng@bc.edu

the framework due to its desirable programming and computation features.¹ While the paper focuses on the sovereign default model of Arellano (2008), the coding designs are relevant for researchers who want to solve complex dynamic stochastic general equilibrium models using parallelization.²

To frame the paper’s contributions, we first give a brief characterization of computing a sovereign default model. This class of models is highly nonlinear and cannot be solved with fast methods like perturbation. A common approach is to use recursive procedures such as value function iteration. However, these methods are slow and suffer from the curse of dimensionality. Value function iteration presents many of the characteristic features of large-scale economic computation: *expensive iterations and frequent matrix storage and access*. These disadvantages make efficient computation of the sovereign default model on a very dense grid attractive. Using C++ and CUDA provides fast computation (Guerron-Quintana, 2016), but at the cost of a steep learning curve and time-consuming coding and debugging. The trade-off is far from ideal for researchers who wish to do complex simulations, but who also want to allocate time away from the technicalities of a low-level programming language.

The paper makes two contributions to address the drawbacks described above. First, we demonstrate Julia’s comparative advantage through testing multiple hardware platforms and languages. The sovereign default model is solved using C++, standard parallel C++, Julia, and Julia with CUDA.³ The spectrum of implementations provides a comprehensive comparison of the advantages and limits of each approach. Julia and Julia CUDA stand out from the comparison due to their excellent trade-off between quick execution speed/high performance and low programming barriers. In most cases, the standard Julia code runs as fast as the C++ code.

Second, we provide an implementation of the sovereign default model based on Julia and Julia with CUDA syntax. Best practices of using Julia are described, which can speed up the computation of the sovereign default model by a factor of 1,000 with GPUs.⁴ Julia demonstrates an exceptional balance in execution speed and ease of development for macroeconomic models. The choice of programming language and GPU compiler reflects our preference (another popular platform is PyTorch with Python). The CUDA toolbox in Julia is an actively developing library, albeit with incomplete implementation

¹<https://julialang.org/>. As of June 2022, Julia 1.7.3 is the stable release. The website <https://julia.quantecon.org/intro.html> provides an excellent introduction of Julia for economists.

²This paper is intended for economists who want to rapidly code and test economic models. It does not seek to provide a comprehensive discussion of coding intricacies.

³Julia with CUDA is available at <https://cuda.juliagpu.org/stable/>. It can be installed from Julia by using `] add CUDA`.

⁴For different language implementations, check the companion GitHub site: <https://github.com/dengmz/ParallelDefault>

compared to CUDA in C (Besard et al., 2018).⁵ The high-level Julia language and the Julia-style design of the CUDA library in Julia provide an efficient infrastructure for testing models on standard Julia and then parallelizing them in Julia using CUDA.⁶

Due to its high learning curve, the use of GPUs in macroeconomics and more generally in economics has been limited. Aldrich et al. (2011) showed the potential of GPUs by solving the neoclassical growth model using CUDA and C. Guerron-Quintana (2016) applied some of their insights to solve the canonical default model using C++ and Thrust.⁷ More recently, Guerron-Quintana et al. (2021) demonstrated that the C++/Thrust infrastructure can be used to speed up the estimation of nonlinear factor models using particle filtering. Gordon and Guerron-Quintana (2019) solve a sovereign default model using Fortran and then use C++/Thrust to extract filtered paths for productivity implied by the model. Finally, Khazanov (2021) uses C++/Thrust to solve a sovereign default model to study currency returns in emerging economies, resulting in a speedup 10 times faster than C++.

Our work is related to Aruoba and Fernandez-Villaverde (2015), who solve the real business cycle model in different languages. Unlike them, we show the algorithmic implementation of the solution of the highly nonlinear default model and use Julia with GPUs. We are close to Fernandez-Villaverde and Zarruk (2018), who provide a practical introduction to parallel computing in economics using different languages, including Julia. They solve a canonical life-cycle model, which is highly amenable to parallelization. Their implementation on GPUs uses CUDA and OpenACC. We view our work as complementary because 1) we show how to use Julia and CUDA to solve highly nonlinear and hard-to-parallelize models; 2) we provide forensic analysis of what drives the computational cost; and 3) we introduce the reader to the novel CUDA toolbox in Julia. Finally, Hatchondo et al. (2010) study the impact of alternative bond grid choices on the accuracy of the solution of sovereign default models. Their focus is on the serial implementation of the solution.

The paper is organized as follows. In Section 2, the noteworthy operations and syntax are explained with examples (assuming that readers have some coding experience with popular coding languages). Section 3 revisits the sovereign default model and the solution algorithm. Section 4 walks the reader through the coding of the solution in Julia with CUDA with an emphasis on tools described in Section 2. In Section 5, we report the significant speed ups through benchmark results. Some practical coding advice is provided

⁵The latest stable version is 3.2.1.

⁶For an excellent introduction to parallel computing, we refer the reader to Kirk and Hwu (2013). Aldrich (2014) provides a gentle introduction to GPU computing for economists.

⁷Thrust is a library of parallel algorithms that tries to replicate the Standard Template Library in C++. It provides a higher level of abstraction compared with CUDA. Recently, NVIDIA released CUB, which is a lower-level library than Thrust to interact with CUDA.

in Section 6. Section 7 concludes.

2 Operations and Syntax in Julia CUDA

This section introduces the Julia CUDA toolbox used to solve the sovereign default model. We give a quick overview of the CUDA universe, but we skip over the low-level CUDA details.⁸ Instead, we focus on a major challenge for economists: how to calculate highly complicated economic models that are expensive to compute on a CPU with the least amount of coding effort? To this end, this section introduces selected CUDA basics to help economists encapsulate complex economic problems into CUDA programming, specifically through kernel programming.

2.1 NVIDIA CUDA in Julia

NVIDIA CUDA in Julia (Julia CUDA) is part of the programming platform of JuliaGPU, which aims to unify GPU programming in Julia. Thanks to its high-level syntax and flexible compiler, Julia is well-positioned to program hardware accelerators like GPUs without losing performance. Indeed, Julia CUDA's simplicity and flexibility in coding are perhaps the most compelling arguments for economists to switch from the original C++ CUDA ecosystem.

Julia CUDA is the best-supported GPU platform among the JuliaGPU platforms (Innes (2020)). It is built on the CUDA toolkit and aims to offer the same level of performance as CUDA C. In Julia CUDA, the CUDA.jl package provides the programming support for using NVIDIA GPUs. Development started in 2014, and CUDA.jl's toolchain is currently mature and can be installed on any current version of Julia using the integrated package manager (Innes (2020)).

CUDA.jl allows programming NVIDIA GPUs at different abstraction levels:

1. By using the CuArray type, the researcher can write powerful abstraction that does not require any GPU programming experience;
2. By writing Julia CUDA kernels, the researcher can compute with speed on a par with kernels written in CUDA C; and
3. By interfacing with CUDA APIs and libraries directly, the researchers can write with the same level of flexibility they would expect from a C-based programming

⁸We recommend in-depth and thorough internet tutorials to interested readers. For Julia CUDA, see <https://juliagpu.gitlab.io/CUDA.jl/tutorials/introduction/>. For C++ CUDA, see <https://developer.nvidia.com/blog/even-easier-introduction-cuda/> and <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

environment (Besard (2016)).

This section focuses on the first two levels of abstractions. We will start with high-level commands such as loop fusion and MapReduce. The CuArray and CUDA kernels are introduced next, bringing us to the second level of abstraction. Kernel programming, in particular, offers more coding flexibility, gentle learning curves, and easier testing when programming complex economic models.

2.2 Loop Fusion

Loop fusion provides access to the convenient Julia CUDA’s linear algebra toolkit, with speed on a par with the well-known CUBLAS package for linear algebra operations in C++ CUDA. Rather than creating typical vectorized loops, loop fusion enables faster calculation with no overhead. This is highly beneficial when working with massive arrays and big data. The overhead in such a scenario could be along the lines of gigabytes, and execution could fail due to memory shortage. Consider a one-million-element matrix for instance: loop fusion offers performance about $4\times$ to $5\times$ faster than separate loops for each computation (Johnson, 2017). In our implementation of utility calculation, we demonstrate how loop fusion reduces execution time from hours by standard broadcast to seconds.

For a simple demonstration, consider evaluating $f(3 * \sqrt{X} + 4X^3)$

```
1 X .= f.(3.*sqrt.(X) + 4.*X.^3)
```

or equivalently

```
1 @. X = f(3*sqrt(X) + 4*X^3)
```

Multiple vectorized operations, such as `sqrt(X)` and `*`, are fused into a single loop with no extra space requirement for temporary arrays. Among scientific programming languages, Julia is unique in offering the loop fusion feature. Other popular languages such as Python or Matlab only allow small sets of operations to be fused, but Julia allows generic application even for user-defined array types and functions. When implementing the value of default in the sovereign model, we will inspect the use and limits of loop fusion.

Loop fusion is, however, most helpful to economists by saving coding effort. As will be demonstrated in Subsection 4.1, loop fusion is a powerful complement to kernel programming, which will be introduced in Subsection 2.5. Kernel programming requires additional time to design and test the kernels. Loop fusion is a good way to efficiently write code that resembles mathematical formulas in the model, thus allowing quick testing of the model and convenience in debugging.

2.3 CuArray

Julia CUDA implements the primary interface of data management through `CuArray`. `CuArray` follows many characteristics of the standard Julia array. A brief introduction to the syntax is referenced below from Julia CUDA’s official documentation (Besard (2016)).

```
1 # generate some data on the CPU
2 cpu = rand{Float32, 1024}
3
4 # allocate on the GPU
5 gpu = CuArray{Float32}(undef, 1024)
6
7 # copy from the CPU to the GPU
8 copyto!(gpu, cpu)
9
10 # download and verify
11 @test cpu == Array(gpu)
```

Code: constructing CuArray

A more concise approach to accomplish these operations is to call the `copy constructor`, i.e. `CuArray(cpu)` (Besard (2016)).

2.4 MapReduce

MapReduce is a widely used method to process large data sets. Google introduced MapReduce in the seminal paper “MapReduce: Simplified Data Processing on Large Clusters” (Dean and Ghemawat, 2008). MapReduce relieves the technicality of programming parallelization and optimization by providing a simple interface. In Julia CUDA, the commands `map` and `reduce` implement MapReduce. Julia CUDA’s MapReduce is highly extensible and can be applied to all types of arrays. In particular, the data structure of interest here, `CuArray`, the standard Julia CUDA array type, works well with MapReduce (Besard et al., 2018).

A standard example is the use of operations `sum` and `max` to compute the expectation. To calculate the expected value $\mathbf{E}[f(y)|y_-]$, one implements

$$\mathbf{E}[f(y)|y_-] = \sum_{y \in \mathcal{Y}} f(y) * Prob(y|y_-)$$

. Assume a vector of endowment states $\mathcal{Y} = \mathcal{Y}_- = (y_1, \dots, y_{N_y})$ of length N_y . The variable `matrix` is an $N_y \times N_y$ matrix with row i repeating endowment state $y_i \in \mathcal{Y}$ for N_y times. The variable `P` is the probability matrix $Prob(y|y_-)$ indexed by $\{y, y_-\} \in \mathcal{Y} \times \mathcal{Y}_-$, so $P[i, j] = Prob(y_i|y_j)$.

A combination of `reduce` and `sum` operates on the matrix and produces the result within a single line:

```
1 Expected_value = reduce(+, reduce(+, f.(matrix).*P, dims=2))
```

For now, ignore the outer `reduce()` and focus on the inner `reduce()`. The second argument of `reduce()` is a set of broadcasting operations that compute the vector of $f(y) * Prob(y|y_-)$. Consider the result as a temporary $N_y \times N_y$ matrix, with value $f(y_i) * Prob(y_i|y_j)$ at position (i, j) . Then, the first and third arguments of `reduce` specify `summing the second argument on the row dimension`,⁹ thus getting a vector of $\mathbf{E}[f(y_i)|y_-]$ for $y_i \in \mathcal{Y}$.

The outer reduce operation would then sum the vector of $\mathbf{E}[f(y_i)|y_-]$ to the desired expected value $\mathbf{E}[f(y)|y_-]$.

The computation can also be further simplified by reducing along all dimensions by default:

```
1 Expected_value = reduce(+, f.(matrix).*P)
```

While the example may seem trivial, note that we have not specified the data structure of variables `matrix` and `P`. This is because the same code will run on both CPU and GPU. At compilation, the compiler will determine if the variables are standard Julia Arrays or Julia CUDA CuArrays. If it is the latter, Julia CUDA will automatically run the code on a GPU, achieving a substantial speedup over CPU computation. And for this speedup, we do not need to write any extra code from the CPU implementation, other than assigning the array as CuArray on a GPU. Thus, in practice, we recommend using MapReduce to autonomize high-level instructions applied to GPU CuArrays for coding efficiency and runtime efficiency.

2.5 Kernels

Kernel programming is the workhorse for our algorithm due to its speed and flexibility. Julia on the CPU is known for computational speed comparable to C, and the same holds for Julia CUDA kernels written using CUDA.jl. The Julia CUDA's official benchmark demonstrates how specifically designed Julia kernels are on a par with or even exceeding the performance of CUDA C kernels (Besard et al., 2018).

Designing good kernels is helpful when implementing difficult economics problems. It's simple to convert economics problems running on normal Julia into a set of kernel functions executable on Julia CUDA, thanks to the well-aligned syntax of standard Julia and Julia CUDA. The use of kernels to speed up the sovereign default model implementation in Julia CUDA is fundamental to our optimization.

⁹Since Julia uses column-major ordering, the parameter `dims=2` indicates reducing by row.

This subsection is organized into the following parts: we begin with a brief hardware-level overview of the GPU architecture, and then introduce basic kernel programming in Julia CUDA. Finally, we construct an example kernel function that increments all elements of a matrix by one.

We start with a quick overview of GPU hardware design to provide helpful intuition about designing kernels. A processor core is the fundamental unit of computing in a GPU. Each processor core is a stream processor capable of running instruction for one thread at a time. In a GPU, processor cores are organized into stream multiprocessors. Figure 1 shows the compositions of stream multiprocessors (SM) in a GPU of the old Fermi architecture and the newer Kepler architecture (Letendre (2013)).



(a) One stream multiprocessor in the Fermi architecture. (b) One stream multiprocessor in the Kepler architecture.

Figure 1: Comparison of Fermi and Kepler stream multiprocessors

Theoretically, CUDA identifies the processor cores through an index system of threads and blocks in a grid. Threads, the virtual representation of processor cores, are piled into thread blocks. The thread blocks are then piled into a single grid.

The size and dimension shared by all thread blocks are customary. A thread block can be one-, two- or three-dimensional. The grid follows the same design. For example, the organization of a grid containing two-dimensional blocks of two-dimensional threads is presented in Figure 2 (Innes, 2017).

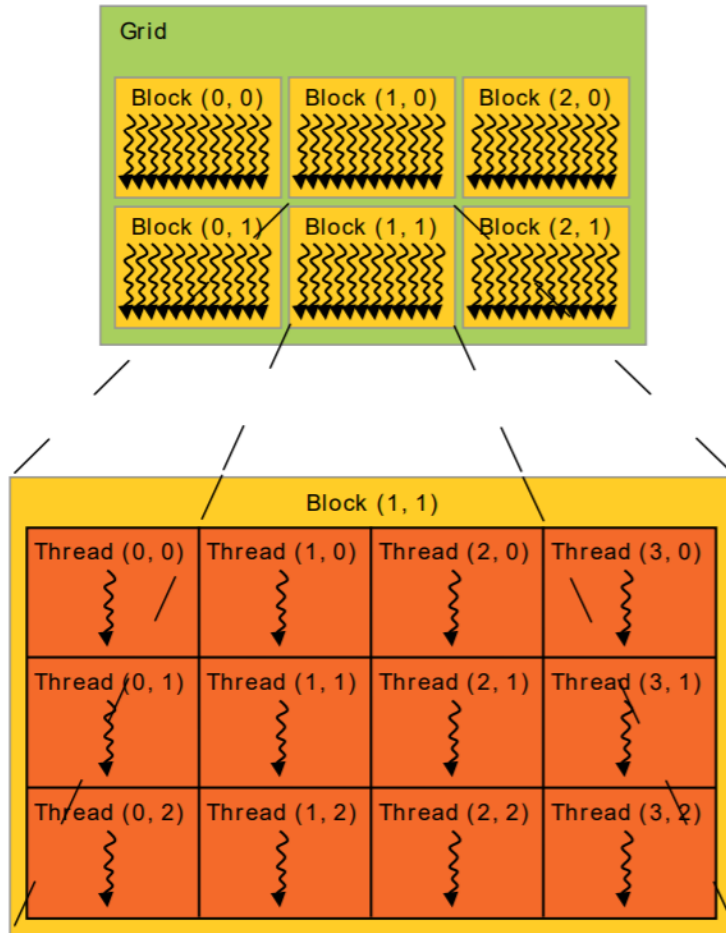


Figure 2: Grid of thread blocks

The following snippet creates a two-dimensional grid containing two-dimensional thread blocks:

```
1 threadcount = (16,16) #two-dimensional thread blocks of size 16*16
2 blockcount  = (16,16) #a two-dimensional grid of size 16*16
```

Revisiting the hardware perspective, all threads of a thread block must be physically allocated in the same stream multiprocessor, in which they share the local memory resource. Currently, each stream multiprocessor contains up to 1024 processor cores by NVIDIA GPU architecture. Therefore, it is a good practice to consider hardware limits when parameterizing a thread block's size. A 16×16 thread block is a good option to avoid inefficiency of thread blocks with too few threads and to avoid the allocation issues with an oversized thread block (keeping threads per block below 512 is good practice).

Once the grids and blocks are customized, we can index the threads to assign different tasks to different threads. This is done in Julia CUDA through two commands: `threadIdx` and `blockDim`. `threadIdx` uniquely identifies the position of a thread within the thread block, and `blockDim` identifies the position of a thread block (which contains

the thread we want) within the grid.

Within the thread block, `threadIdx` provides the Cartesian indexing of the relative position of a thread. `threadIdx` provides the unique x, y and z coordinates (or fewer coordinates depending on the dimension of the thread block). The same argument follows for indexing a thread block within a grid using the command `blockDim`.

Now we introduce a standard practice of assigning a loop in the algorithm to a dimension in the thread-block-grid structure. For example, to compute a double for-loop where each element in matrix M is incremented by 1:

```
1 for x in 1:Nx
2     for y in 1:Ny
3         M[x,y] += 1
4     end
5 end
```

we can organize the threads into a two-dimensional matrix. Each thread is indexed and assigned to position (x, y) in a double for-loop. The kernel then calculates the output at (x, y) .

The following command demonstrates how to calculate the index of each thread:

```
1 x = (blockIdx().x-1)*blockDim().x + threadIdx().x
2 y = (blockIdx().y-1)*blockDim().y + threadIdx().y
```

In contrast to C++ CUDA indexing, indices of `blockIdx` are each decremented by 1, since Julia uses 1-indexing instead of C's 0-indexing.

We want each kernel to calculate a feasible state in the grid, so execution at a particular grid point naturally requires passing a conditional statement. In this case, we want the range to be bounded by $[1, N_x]$ and $[1, N_y]$:

```
1 if ( x <= Nx && y <= Ny)
```

Combining the identification, condition of execution, and execution elements, we have a complete kernel. Note that we feed loop sizes `Nx` and `Ny` as variables into the kernel function:

```

1 function example(M,Nx,Ny)
2     x = (blockIdx().x-1)*blockDim().x + threadIdx().x
3     y = (blockIdx().y-1)*blockDim().y + threadIdx().y
4
5     if ( x <= Nx && y <= Ny)
6         M[x,y] += 1
7     end
8
9     return # a return statement is necessary at the end of a kernel
10 end

```

Code: example kernel

We highlight the ease of modifying the original double-loop in standard Julia to a kernel function in Julia CUDA. After writing the prototype code in Julia, the entire effort lies in designing the dimensions of threads. Advanced features such as memory allocation will be addressed when discussing coding practices. However, one of our main messages is the power of simplicity with Julia CUDA: vanilla kernel programming provides satisfactory computation power for complex and granular economic models.¹⁰

After finishing the kernel design, we proceed to execute the kernel. First, we assign the sizes of the thread blocks and the grid. We pick the natural choice of the size (16, 16) of the thread blocks. The size of the grid is the smallest grid of size over (N_x, N_y) that is covered by size (16, 16) blocks. The third line demonstrates how to execute the kernel by providing the threadcount and blockcount parameters:

```

1 #Construct a grid of size (Nb threads) * (Ny threads)
2 threadcount = (16,16)
3 blockcount  = (ceil(Int,Nb/16),ceil(Int,Ny/16))
4 @cuda threads=threadcount blocks=blockcount example(M,Nx,Ny)

```

The sample code is provided in the appendix.

3 Sovereign Default Model

We work with the standard sovereign default model (Arellano, 2008). Here, we provide a brief description of the model (we refer the reader to the original paper for details). The model contains an open economy with a central government. The small open economy receives a stochastic endowment y each period. The law of motion is

$$y' = \rho y + \sigma_y \epsilon, \quad \epsilon \sim N(0, 1).$$

¹⁰Kernels in Julia CUDA resemble the functor construct heavily used in C++ and CUDA programming.

Each period, the government chooses between repaying its debt obligations or defaulting payment to maximize households' utility. The value-of-repayment is

$$V_r(y, b) = \max_{b' \leq 0, c \geq 0} U(c) + \beta \mathbf{E}V(y', b')$$

subject to the budget constraint

$$c + q(y, b')b' \leq y + b,$$

where $q(y, b')$ is the price of debt issued today, b' , given that the endowment is y . \mathbf{E} is the expectation operator over future shocks.

The value function if defaulting is given by

$$V_d(y) = U((1 - \tau)y) + \mathbf{E}(\phi V(y', 0) + (1 - \phi)V_r(y')).$$

Here τ is the fraction of the endowment lost because of default and ϕ is the exogenous probability of being readmitted to the financial markets next period. Note, if readmitted, the sovereign economy starts with zero liabilities. The planner selects a default choice d by solving the problem $V(y, b) = \max_{d \in \{0, 1\}} (dV_d(y) + (1 - d)V_r(y, b))$.

It is worth noting the two computation roadblocks that the solution of the model presents. First, the solution is highly nonlinear due to the max operator involved in the planner's decision of whether to default or not. Second, the price of debt issued today $q(y, b')$ depends on the probability that the country defaults tomorrow, which is an endogenous object. That is, today's actions depends on what the sovereign economy does tomorrow. But its actions tomorrow depend on how much debt she chooses today. Together, these two points rule out solution methods based on perturbations or projections. Value function iteration is the only viable method.

In our implementation, we follow Guerron-Quintana's (2016) parameterization:

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

where $\beta = 0.953, \rho = 0.9, \sigma_y = 0.025, \tau = 0.15, \phi = 0.28$. The endowment process is discretized using Tauchen's (1986) method.

The algorithm to solve the model discretizes the support of the states, resulting in a grid of points in the space of endowment and bonds $(\mathcal{Y} \times \mathcal{B})$. For each grid point and conditional on the price of debt computed in the previous iteration, the values of default and repayment are sequentially calculated. When computing the repayment value, the researcher needs to check whether the choice leads to positive consumption. A decision is

made for repayment or default for each grid point, and the price of debt for the current iteration is updated. With this update in hand, one proceeds to compute the values of default and repayment at each grid point. We continue with this updating process until consecutive iterations render value functions within a predefined error tolerance.

4 The Sovereign Default Model in Julia

This section describes our main contribution – the parallel computation of the sovereign default model in Julia CUDA. We divide the implementation into three computation-heavy components: value-of-default, value-of-repayment, and decision. Our design is based on implementation through Thrust in C CUDA. We hope to demonstrate how the Julia environment helps economists code simply, yet effectively.

The components were chosen to emphasize two key features of Julia CUDA: flexibility and simplicity. The value-of-default section serves as a sandbox for experimenting with the various tools outlined in Section 2. Meanwhile, the value-of-repayment part focuses on a simple kernel programming application. In Section 5, we will demonstrate the tremendous performance speedup from the simple Julia CUDA model we will build in this section.

4.1 Value-of-Default

Reduce operation: In the default model, the expected value calculation is implemented by $\mathbf{E}[f(y)|iy] = \sum_{y \in \mathcal{Y}} f(y) * P(y|iy)$. To compute $\mathbf{E}[f(y)|iy]$ for each pair of (y, iy) , one common method is to run through a double for-loop through each state of $y \times iy$:

```

1 sum_default = CUDA.zeros(Ny)
2 # Ny is the size of grid points of possible values of y
3 for y in 1:Ny
4     for iy in 1:Ny
5         sumdefault[y] += f(y)*P[y,iy]
6     end
7 end

```

Temporary matrix: An alternative to a double loop is to store $f(y|iy)$ for each pair of possible values for (y, iy) in a temporarily initialized matrix and sum along the columns.

```

1 #Initialize temporary matrix
2 temp_vd = CUDA.zeros(Ny,Ny)
3
4 for y in 1:Ny
5     for iy in 1:Ny
6         temp_vd[y,iy] = f(y)*P[y,iy]

```

```

7     end
8 end
9 sum_default = reduce(+, temp_vd, dims=2)

```

Code: store and reduce with a temporary matrix

Loop fusion: Given the value function

$$f(y) = \phi V(y', 0) + (1 - \phi) V_d(y') | y(i_y),$$

an alternative method is to use loop fusion. Recall that loop fusion provides a significant speedup by just *adding a few dots*, thus keeping the form of math formula *intact*. In this example, loop fusion essentially fuses Julia's primitive linear algebra calculation to broadcast the operation across each element of the matrix. The following snippet is equivalent to $f(y)$:

```

1 phi.*V[y',0] .+ (1-phi).*V_d[y',i_y]

```

And thus we can summarize the value-of-default calculation in two lines:

```

1 temp = beta * P .* CUDA.transpose(phi.*V[y',0] .+
2     (1-phi).*V_d[y',i_y])
3 Vd .= sumdef .+ reduce(+, temp, dims=2)

```

4.2 Value-of-Repayment

Value-of-repayment consumes the largest bulk of computation power and is the priority for optimization. Our implementation of the value-of-repayment component provides the greatest speedup. The computational cost comes from an expected value calculation that loops through four variables, $\{ib, iy, b, y\}$, and is of complexity $O(n^4)$.¹¹ In our CPU design, Julia-style linear algebra and loop-fusion operations again replace the standardized for-loops.

```

1 for ib in 1:Nb
2     for iy = 1:Ny
3         Max = -Inf
4         for b in 1:Nb
5             c = exp(Y[iy]) + B[ib] - Price0[iy,b]*B[b]
6             if c > 0
7                 sumret = 0
8                 for y in 1:Ny

```

¹¹Since the Julia ' operator is currently unsupported by the Julia CUDA package for kernel computation, the calculation of the sum-of-return requires manually implementing one additional for-loop.

```

9         sumret += P[iy,y]*V0[y,b]
10     end
11     vr = U(c) + beta * sumret
12     Max = max(Max, vr)
13 end
14 end
15 Vr[iy,ib] = Max
16 end
17 end

```

Code: Julia CPU code for the value-of-repayment algorithm

In Guerron-Quintana’s (2016) CUDA implementation, the Thrust library provides an efficient and intuitive transition from CPU code to GPU code. This section shows how we extend and improve upon the Thrust design. First, we inspect the implicit extra computation cost in the Thrust implementation and address the synchronization problem. The problem occurs because the large variation in execution time among the sizable threads may greatly impact overall performance. During every round of computation, the faster threads will wait for the slower threads to finish the calculation, requiring additional synchronization time for each round of computation. In addition, some identical data will be repeatedly calculated and freshly stored on the device for each thread, requiring additional device space and computation power.

Next we assign the dimensions of the thread blocks and the grid to fit the structure of the value-of-repayment problem. We can encapsulate a for-loop into a dimension of the thread block or the grid. And recall that the value-of-repayment component produces the value of repaying the debt for each pair in the endowment \times bond grid.

Therefore, we design a two-dimensional thread block where each thread computes the value of repayment for a particular pair of states (iy,ib).

Encapsulating all four loops through a two-dimensional thread block and a two-dimensional grid may be a tempting choice, as it minimizes runtime per kernel. However, the memory transfer and storage conditions are too expensive. Each thread requires the price matrix and the original value matrix from the GPU memory: `sumret += P[iy,y]*V0[y,b]`. Suppose we parametrized an endowment \times debt grid with $N_y = 500$, $N_b = 500$, and thread blocks of size (16, 16). As a thread block shares the local memory, a total of about $2 \times 500^2 \times 500^2 / 16^2 = 768,528$ transfers of floats will be required just to begin the calculation.

Instead, we designate each thread to perform the computation of the two inner loops of the value-of-repayment algorithm. In other words, we assign each thread to compute the value-of-repayment value for a particular pair in the endowment \times bond grid.

```

1 function vr(Nb,Ny,alpha,beta,tau,Vr,V0,Y,B,Price0,P)
2
3     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
4     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
5
6     if (ib <= Nb && iy <= Ny)
7
8         Max = -Inf
9         for b in 1:Nb
10             c = CUDA.exp(Y[iy]) + B[ib] - Price0[iy,b]*B[b]
11             if c > 0 #If consumption positive, calculate value of
return
12                 sumret = 0
13                 for y in 1:Ny
14                     sumret += V0[y,b]*P[iy,y]
15                 end
16
17                 vr = CUDA.pow(c,(1-alpha))/(1-alpha) + beta * sumret
18                 Max = CUDA.max(Max, vr)
19             end
20         end
21         Vr[iy,ib] = Max
22     end
23     return
24 end

```

Code: kernel of value-of-repayment component

We readdress the similarity between the Julia CPU code and the Julia CUDA code. The only modification is to encapsulate the outer two for-loops of ib and iy into the dimension of the thread block. Such a design reduces the complexity from $O(n^4)$ to roughly $O(n^2)$. The benchmark result will showcase the power of a simple kernel: GPU speeds up the value-of-repayment computation by 1,000 times compared to standard Julia.

5 Benchmarking

The benchmark is tested on an Intel Core i7-10750H CPU @ 2.60GHz with NVIDIA GeForce RTX 2060. The Julia version is 1.4.2 (2020-05-23) and the CUDA library is v0.1.0.¹²

¹²The code was also tested on an AERO-Gigabyte laptop with Intel Core i9-1098HK @ 2.40GHz, Nvidia GeForce RTX 3080 running Ubuntu 20.04 and Julia 1.5.

5.1 Julia CUDA

The benchmark analysis is performed on the three major computation components: value-of-default, value-of-repayment, and decision. Figure 3 plots the results from the Julia CUDA implementation. The X-axis depicts the size of grid points for the endowment \times bond matrix of size $N_y \times N_b$, and the Y-axis measures the median running time in microseconds for one evaluation of each component of the model.

Our analysis focuses on two parameterizations with different emphasis. In this subsection, we introduce the first benchmark, which aims to maximize the difference in complexity magnitude among the components. For simplicity and clarity, we will assign an equal number of endowment and bond points, $n = N_y = N_b$. The endowment \times bond grid is then a square matrix containing n possible endowment values and n possible bond values. The value-of-repayment component is of complexity $O(n^4)$, and the value-of-default and decision components are of complexity $O(n^2)$. The value-of-repayment thus dominates runtime, and Julia CUDA will deliver the most significant speedup in this component.

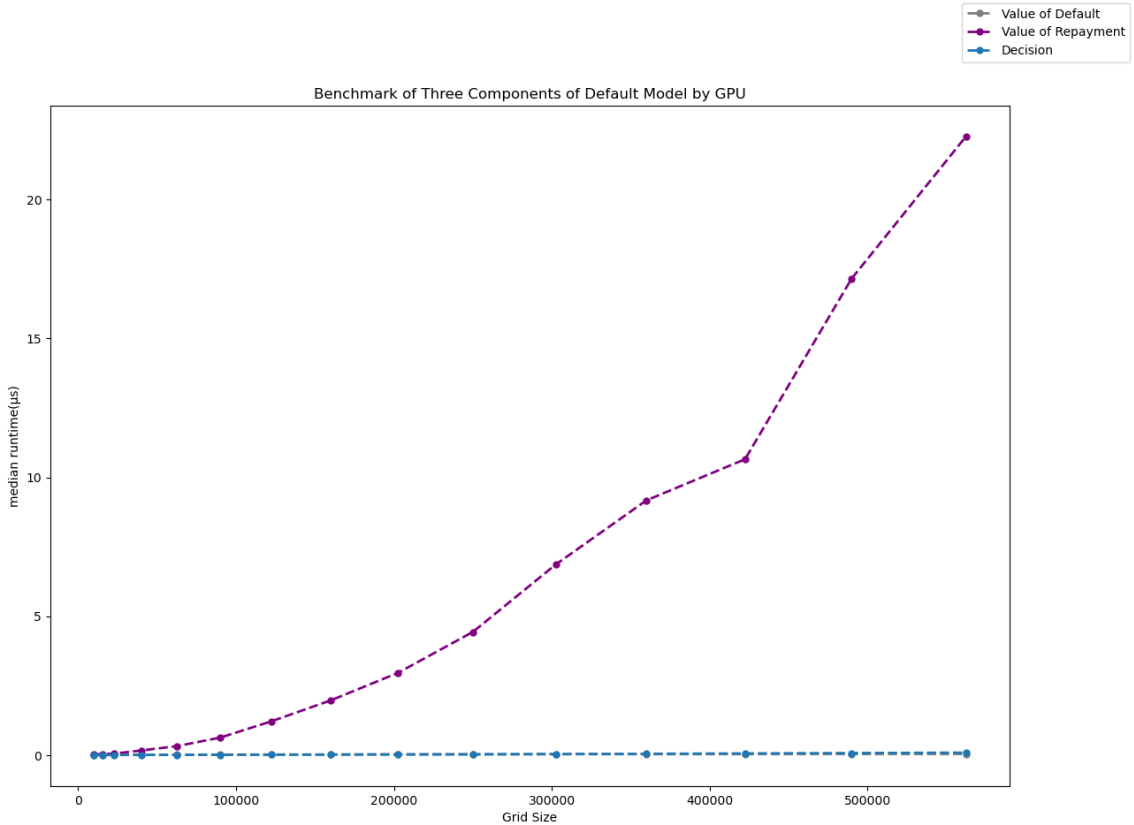


Figure 3: GPU Benchmark of sovereign default model

Figure 3 showcases the performance under grid granulation from 100×100 up to 800×800 grid points for the endowment \times bond matrix. Note the value-of-default line

practically overlaps with the decision line in the graph, again demonstrating how the value-of-repayment component dominates runtime.

A mild polynomial trend may be seen for the value of repayment component, which has complexity $O(n^4)$. Even with a vast grid, the GPU processor has not hit its computational limit. The computation is quick, with each round of Value-of-Repayment calculation taking less than two seconds on a very granularized grid of size 160,000.

A major bottleneck to our benchmark comes from memory usage. For example, to store a single cost matrix with $n = 500$ requires 500^3 of Float32 memory, equivalent to $500^3 \times 32/10^9 = 4\text{GB}$ of RAM.¹³ At $n = 800$, the required RAM memory reaches 16 GB. Further granulation would be uneconomical on a personal computer.¹⁴

5.2 Comparison: Julia CUDA vs. Julia CPU

The Julia CUDA implementation of the sovereign default model is based on the Julia CPU implementation of the sovereign default model. Due to the close resemblance of the two implementations, the benchmark comparison offers a fair demonstration of the speedup achieved by GPU acceleration.

In this section, we modify the first benchmark to aim for a more realistic setting. Instead of assigning $n = N_y = N_b$, this benchmark follows a more standard parameterization of $N_y = 7$ endowment points with varying debt granularization of $N_b = n$ debt points, with a constant N_y . The value-of-repayment component has complexity $O(n^2)$, the value-of-default is of $O(1)$ and decision have complexity $O(n^2)$.

Figure 4 shows that the implementation in the GPU is faster than the one in the CPU. Note the value-of-repayment component dominates runtime. This component has complexity $O(n^2)$. As the number of grid points increases, the solution in the CPU becomes quickly infeasible. We ignore the value-of-default part given its constant complexity.

¹³Using Float64 on the GPU is prone to error.

¹⁴A quick fix of the memory issue is to remove the high-dimensional matrix that had sped up the calculation. In the trade-off of memory for speed, we replace the standard vectorized loops with MapReduce operations and linear algebra operations on high-dimensional matrices. In the next section of coding practices, we also provide a method to get around the memory limit boundary in GPU hardware for interested readers.

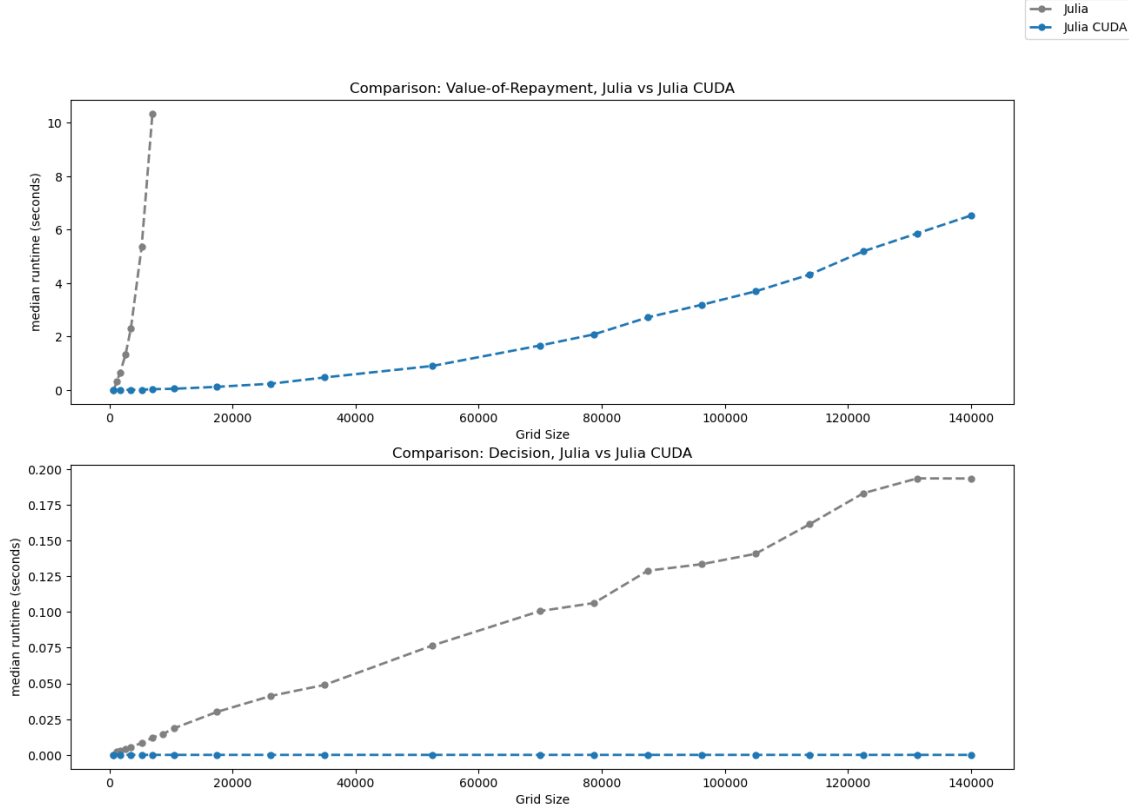


Figure 4: First benchmark of grid size $7 \times n$. Comparison of the last two components in the model

Completing the analysis of the realistic benchmark, we return to the original benchmark with $n = N_y = N_b$. The complexity is again $O(n^4)$ for value-of-repayment, $O(n^2)$ for value-of-default and decision.

Figure 5 shows the dominating performance of the GPU solution given two extra complexity orders. The program’s runtime performance is again dominated by the most time-consuming process: value-of-repayment calculation. For CPU computation, the runtime of the value-of-repayment calculation grows in polynomial scale to over 10 seconds on a 100×100 grid. Suppose the algorithm takes 500 iterations. Then calculating value-of-repayment alone takes over an hour. Meanwhile, Julia CUDA completes the same task in $0.0467s \times 500 = 23$ seconds.¹⁵

¹⁵The drastic slowdown in the (serial) CPU implementation confirms the results in Table 5 in Hatchondo et al. (2010).

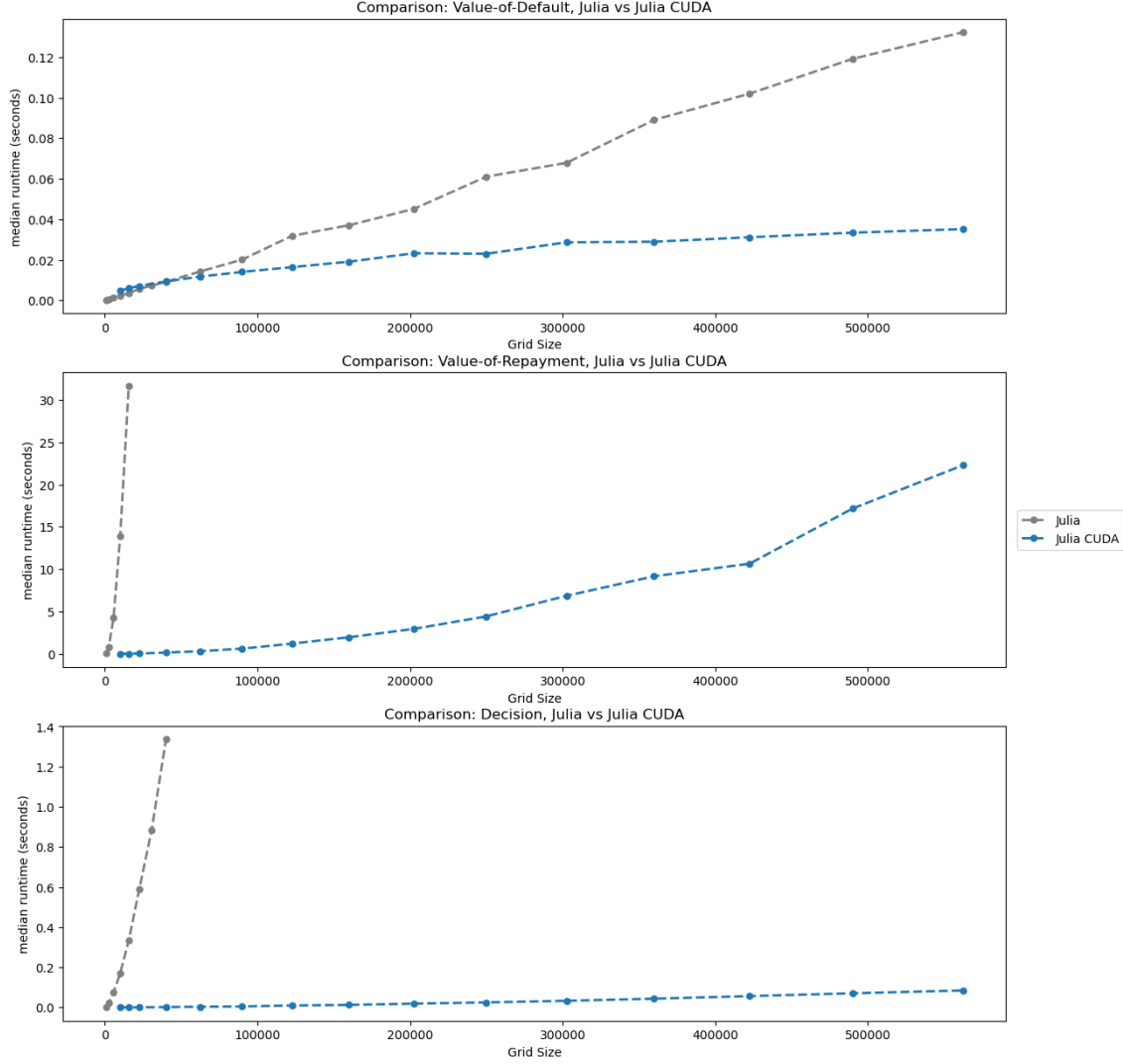


Figure 5: Second benchmark of grid size $n \times n$. Comparison of the three components in the model

5.3 Julia CUDA vs. C++ Stdpar CUDA

We compare Julia CUDA's computation speed to that of the C++ CUDA standard parallel library (Stdpar) implementation in this section. Julia CUDA outperforms C++ Stdpar in the most time-consuming component: value-of-repayment in Figure 6 and Figure 7. This speedup is particularly significant in the second benchmark with grid size $n \times n$ in Figure 7.

It is worth noting that the benchmarked Julia CUDA and Stdpar C++ implementations are not mirror copies in terms of code designs. The C++ Stdpar model implementation aims to circumvent the technical difficulties of programming CUDA with standard C++ code. Using features of object-oriented programming, C++ Stdpar encapsulates all data grids and calculation components in a single class and then processes in C++ Std-

par code to run in CUDA. The design simplifies the code transfer process from standard C++ to C++ Stdpar.

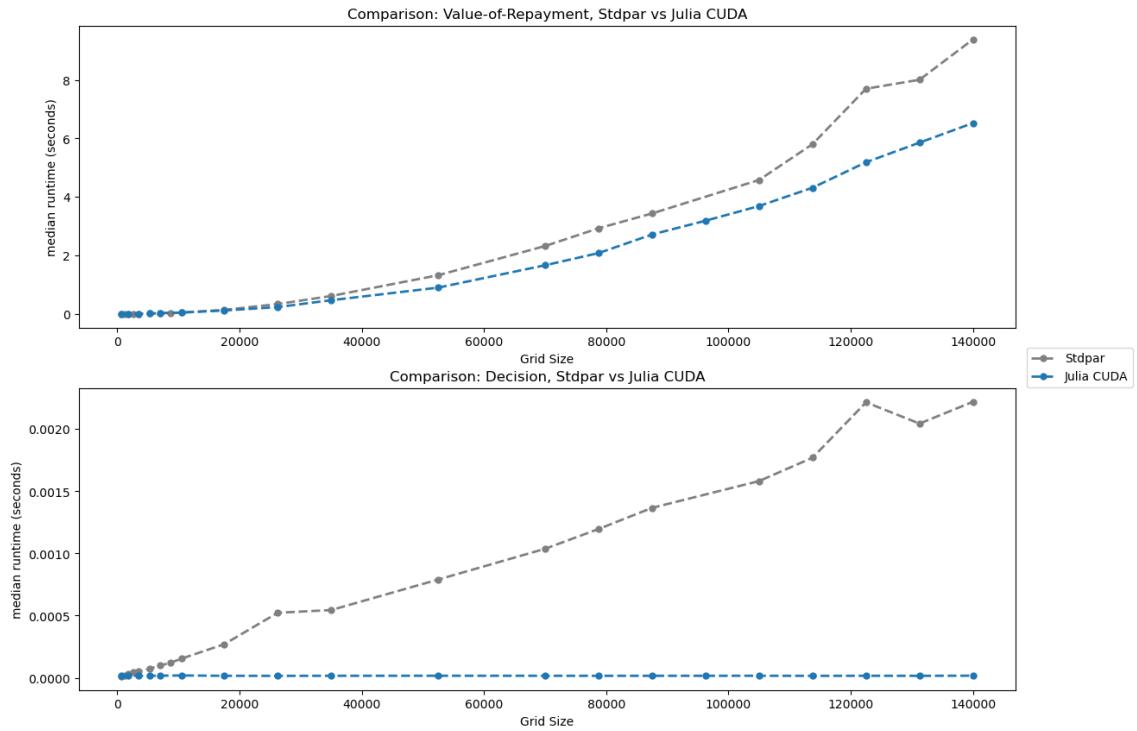


Figure 6: First benchmark of grid size $7 \times n$. Comparison of the last two components in the model

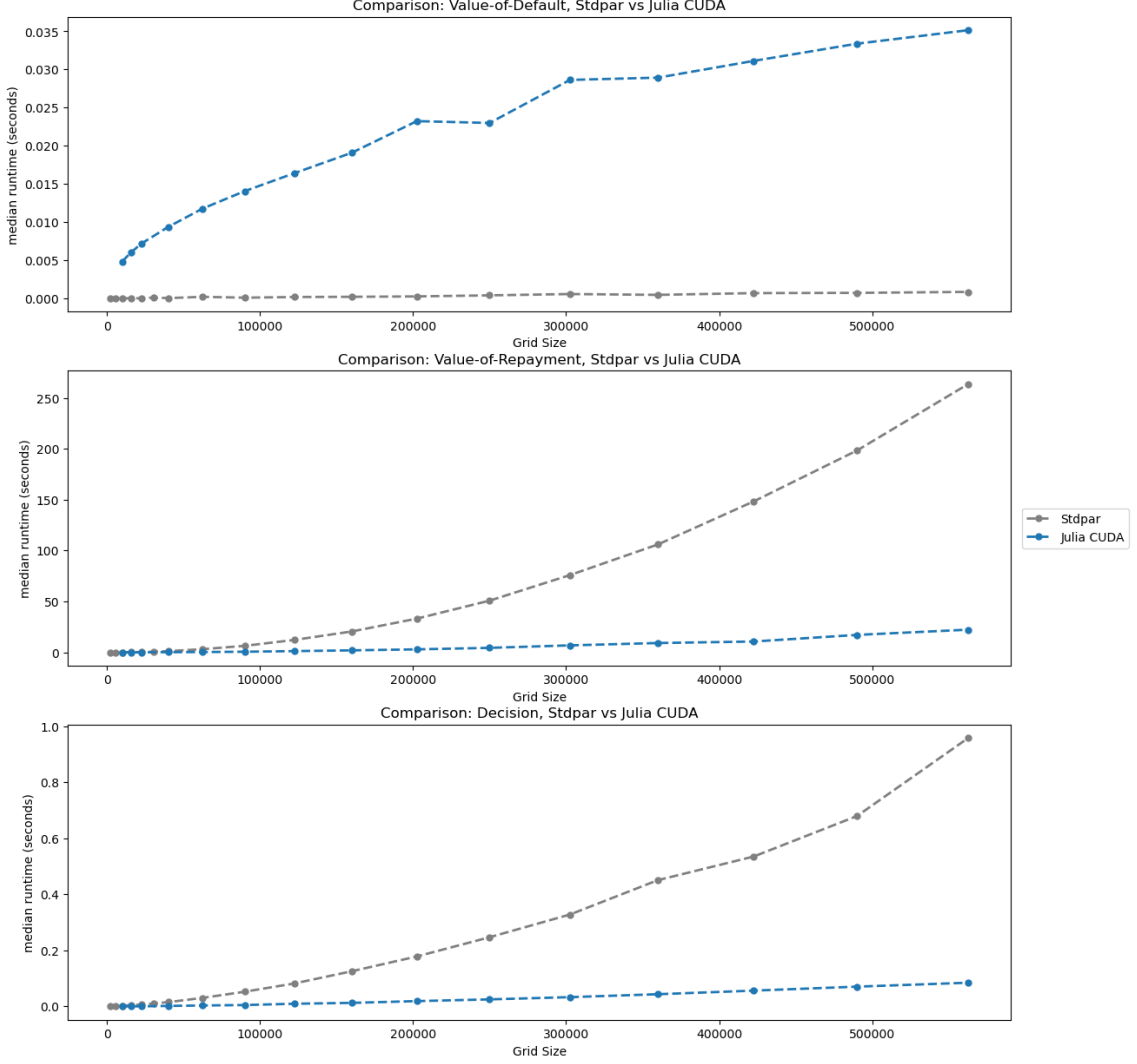


Figure 7: Second benchmark of grid size $n \times n$. Comparison of the three components in the model

Julia CUDA possesses a great advantage in the computation of value-of-repayment and decision, while Stdpar is faster in calculating value-of-default. The next subsection confirms Julia CUDA’s advantage over Stdpar by benchmarking real execution of the entire model.

5.4 Coding Benchmarks in Julia CUDA

Coding the benchmark in Julia CUDA deserves special attention. Correctly measuring the performance was a great challenge given the relatively short history of Julia CUDA. This subsection aims to outline a simple and feasible way to benchmark Julia CUDA with the `BenchmarkTools` library through the command `@benchmark`. We’ve included our many failed benchmark attempts in the appendix.

We present our benchmark method, a combination of `@benchmark` with inner loops.

Instead of directly benchmarking the kernel, we encapsulate the kernel in a new function, which iterates the kernel several times. Applying `@benchmark` directly to a Julia CUDA kernel does not produce the correct result, possibly due to compatibility issues between standard Julia platform and the Julia CUDA package. The objective of the iteration is to minimize the effect of Julia’s special feature, precompilation, which occurs when a piece of code is executed for the first time. Without iteration, precompilation would distort the benchmark with a disproportionate time on overhead. The standard Julia benchmark follows the same design. The following code benchmarks value-of-repayment:

```

1 sec=120 #set time allowance for benchmarking
2 test_rounds = 10 #number of rounds to iter the kernel
3
4 function GPU_VR()
5     for i in 1:test_rounds
6         @cuda threads=threadcount blocks=blockcount vr(Nb,Ny,alpha,beta
7             ,tau,Vr,V0,Y,B,Price0,P)
8     end
9 end
10 t_vr = @benchmark GPU_VR() seconds = sec
11 time_vr = median(t_vr).time/1e9/test_rounds #divide by 1e9 to convert
    to seconds

```

Code: benchmarking the Value-of-Repayment component

6 Coding Practices: The Good, the Bad, and the Ugly

While powerful, CUDA in Julia is still a relatively young platform. Julia 1.0 was launched on August 7, 2018, and is the oldest version still supported. The paper uses Julia 1.4.2 (2020-05-23) – the latest update is version 1.7.3 (May 2022). The ecosystem is thriving, but still relatively small. This section aims to point out some pitfalls to avoid based on our sovereign default model implementation.

6.1 Synchronization

In this subsection, we explain how to dissect the sovereign default model algorithm into multiple kernel functions. We recommend such modular designs for computational-heavy algorithms. The benefits in coding efficiency and in sharing the code design across programming platforms outweighs the labor cost to dissect the algorithm.

Synchronization cost arises when various threads within the GPU need to share data among themselves (Letendre (2013)). The implementation of most economic problems requires a complex control flow that includes communication across threads.

Consider the example of calculating debt price in the sovereign default model implementation. Suppose we write a single kernel function that evaluates the entire algorithm at that endowment \times debt point, to calculate the value-of-default, value-of-repayment, decision, and prices given the particular endowment \times debt index. To derive the debt price at a particular endowment \times debt point, we need to retrieve data from multiple rows and columns of the updated value-of-default and value-of-repayment matrices. Therefore, in real runtime, the matrices of the three components must be fully updated by the respective three kernels before any single price point could be correctly derived.

We recommend partitioning large kernels to improve performance.¹⁶ Such division has multiple benefits. Compared to a single kernel function, runtime variance among kernels is drastically reduced. Also, debugging and feature-adding are much more efficient in a modular design.

However, we warn the readers about the potential costs of dividing kernels. Redundant kernel initialization and memory transfer speed could remove any intended improvement in speed. Fortunately, thanks to CUDA's power, there is often no need to investigate the details of the kernels to speedup the program for our purposes.

6.2 Memory Management through Garbage Collection

To improve the performance of data-heavy economic models running in Julia CUDA, we extend the discussion of CuArray to introduce garbage collection following Julia CUDA's official manual in this section.

Julia CUDA automatically stores user-created objects and caches the underlying memory in a memory pool to speed up future allocations. When memory pressure is high, the memory pool automatically frees the cached objects (Besard (2016)). To free up and reclaim the cached memory, call `CUDA.reclaim()`. Note manual reclaim is not necessary for high-level GPU array operations. Therefore, if the user runs into an out-of-memory situation, as will be discussed in the next subsection on temporary matrix, reclaiming cached memory will not solve the problem.

Since a GPU has a small memory and will frequently run out of it, avoiding automatic garbage collection could significantly improve performance. Calling the `unsafe_free!` method frees up manual memory free up without calling the Julia garbage collector.

The batching iterator provides an interface to input data beyond the memory capacity

¹⁶A good reference is Letendre, J.(2013).

of a GPU with automatic memory free-up. An official example of using `CuIterator` to batch operations is provided below (Besard (2016)):

```
1 batches = [([1],[2]),([3],[4])]
2
3 for (batch, (a,b)) in enumerate(CuIterator(batches))
4     println("Batch $batch: ", a .+ b)
5 end
6
7 Batch 1: [3]
8 Batch 2: [7]
```

Code: iterate and map on the batched data

In the example code, we feed batches of divided data that can each be fitted into the GPU: `batches = [([1],[2]),([3],[4])]` to `CuIterator`. `CuIterator` can then iterate through the batches and operate on each batch on a GPU. The example code performs a matrix addition on the two matrices in each chunk. When loading a new batch onto the GPU, `CuIterator` frees the old batch from the GPU memory through the command `unsafe_free!`. This automatic process removes the need to free up manual memory or to call the Julia garbage collector in a GPU. Note that while easing the pressure on the GPU memory allocation, `CuIterator` is not responsible for speeding up the memory transfer process from CPU to GPU.

`CuIterator` is, however, not helpful for the sovereign default model. In the sovereign default algorithm, computation at each state of $\text{endowment} \times \text{debt}$ requires access to all possible states of $\text{endowment} \times \text{debt}$, and thus access to the entire $\text{endowment} \times \text{debt}$ matrix. Dividing the $\text{endowment} \times \text{debt}$ matrix into batches effectively blocks access to the entire matrix for each batch.

6.3 Temporary Matrix

Allocating a temporary matrix during initialization is a double-edged sword that necessitates close examination. Temporary matrices provide an optional trade-off to increase speed at the cost of extra memory allocation on a GPU.

For instance, the appendix includes the calculation of the value-of-repayment component through temporary matrices. Several temporary matrices were created to facilitate the calculation. For example, the cost matrix $C[iy, ib, b]$ is a temporary matrix containing three variables. Allocating the cost matrix beforehand on the device/GPU removes the dynamic allocation of memory for cost in each thread. The freshly calculated cost will be

assigned to the pre-allocated matrix stored in the respective blocks. Efficient linear algebra operations could consequently be performed on the matrix, offering better speedup than the standard vectorized for-loops.

The greatest benefit of temporary matrices is the ability to divide the problem. Instead of crowding all operations into a single kernel, each step of the calculation could be handled by a kernel, with the output stored for the next step. Another benefit is that temporary matrices could use the built-in functions of Julia CUDA for quick matrix manipulations.

The disadvantages of temporary matrices are memory overflow and transfer inefficiency.¹⁷ First, temporary matrices can be large and can lead to memory overflow. Take the cost matrix $C[iy, ib, b]$ for example. Suppose there are 7 endowment levels with 10,000 debt levels. The size of matrix C then reaches 5.215 GiB. If the cost matrix is stored on GPU memory, a potential overflow could occur. For example, the paper’s benchmark is run on a GeForce RTX 2060 graphics card with 6GB GDDR6 memory. A single 5.215 GiB temporary matrix would be dangerously close to a memory overflow. A natural option is to store the matrix on CPU, and transfer the matrix when computation is needed. This leads to the second disadvantage of data transfer cost.

Data transfer cost can make temporary matrices prohibitively expensive for value-iteration calculations. According to NVIDIA’s best practices guide for CUDA, memory optimization is the most critical area for performance, and maximizing bandwidth is crucial for memory optimization. Data transfer between host and device drastically reduces the theoretical bandwidth, for example, from “898 GiB/s on the NVIDIA Tesla V100” to “16 GiB/s on the PCIe x16 Gen3.” Suppose the matrix is instead stored on the host CPU. For each of the 300 rounds of value iteration, the 5.215 GiB matrix must be transferred from the CPU to the GPU and back again. If the theoretical transfer speed is 16 GiB/s, the transfer cost for a single temporary matrix would already take more than $5.215/16 \times 2 \times 300 = 195$ seconds.¹⁸ With more temporary matrices, the data transfer cost becomes highly undesirable.¹⁹

It is thus preferable to run kernels on the GPU to minimize data transfer to the host CPU. However, this would reintroduce the first disadvantage: the memory is stored on the GPU with limited memory. A dilemma between storing the matrix on the CPU or the GPU thus occurs.

¹⁷Due to these deficiencies, our standard implementation for Julia CUDA does not use temporary matrices.

¹⁸300 is a rough estimation of the number of iterations required to run the sovereign default algorithm with error tolerance $1e-6$.

¹⁹In C++ CUDA, the unified memory model allows better data transfer between CPU and GPU memory. Adding the feature to Julia CUDA could be helpful, but not essential for efficient computation of economic models in Julia CUDA.

6.4 Loop Fusion

Splitting loop fusion into smaller sections may improve performance if no extra memory is allocated. The dots in loop fusion are essentially broadcast operations. The greatest improvement in loop fusion comes by removing the costly allocation and deallocation process in creating temporary arrays. However, automatically fusing multiple broadcast operations does not result in the expected level of improvement. The speedup of loop fusion diminishes when too many loops are fused.

Instead, dividing the fusion back to individual broadcast operations may offer better performance if the code is designed to allocate minimal extra space to store temporary results. For example, the following snippet to calculate $temp = \beta P(A)^T$:

```
1 temp = P
2 temp .*= CUDA.transpose(A)
3 temp .*= beta
```

is much faster than

```
1 temp = beta* P .* CUDA.transpose(A)
```

in Julia and no extra array is allocated.

In practice, we recommend splitting the operation as above for best performance, for reasons similar to those discussed in the Synchronization subsection.

6.5 Limits of Kernel Computation for Economic Models

We point out some “ugly” sides of GPU computation for certain types of economic models to complete the title of the section.²⁰

The complexity of control flows in a model could escalate the challenge in kernel designs and runtime performance. A GPU is most powerful for running the SIMD (single instruction multiple data) operations. CUDA could perform poorly when the model features numerous control flows and thus requires many synchronization checkpoints. The sovereign default model features a relatively easy control flow: there are only two types of conditional statements that could be contained within a single simple kernel. The check on a realistic cost is contained in the value-of-default and the value-of-repayment kernels. Choosing to default or not is a simple statement contained in the decision kernel. There are no recursive iterations or huge divergence in calculation procedures in the sovereign default model. Based on the results in Fernandez-Villaverde and Zarruk (2018) and Guerron-Quintana et al. (2021), our expectation is that Julia CUDA implementations of

²⁰Particular types of computer science problems, such as graph algorithm, searching, and sorting fall into this category. But given their relatively small importance in economics than in computer science, we do not go into details about these interesting fields.

state-of-the art heterogeneous agent models will experience significant improvements in speedup compared to serial implementations.²¹

The limit of the memory transfer bandwidth, as pointed out in the temporary matrix subsection, could slow down runtime performance. If we can not fit all data in the GPU memory, we have to copy the memory from the CPU to the GPU, and feed the updates back from the GPU to the CPU. The runtime transfer cost between the GPU and the CPU could dominate the actual computation cost spent in the GPU. We introduced one method in the memory management subsection to improve the situation: use BatchIterator to perform large-scale data transfer from the CPU to the GPU, and perform manual garbage collection to speed up performance.

Overall, managing and unleashing the full potential of (Julia) CUDA requires examination of programming details. Complexities in control flow and memory transfer require scrutiny and coordination in code designs. However, performance-wise, these challenges are minor compared to the greater benefits of Julia CUDA programming.

7 Concluding Remarks

This paper proposes a framework for parallel processing in Julia, which is motivated by ever more complicated and improved methods in nonlinear quantitative models. The language choice demonstrates our faith in Julia as a growing numerical language. Julia allows for a better mix of execution speed and coding ease. This paper focuses on computational details and coding standards for parallel computation in Julia. Using the sovereign default model as an example, we illustrate the strength of parallel computation in iterative models with knife-edge decision rules. We demonstrate how the Julia environment allows writing simple parallelization codes, thus reducing coding time and execution time in implementing realistic models like sovereign default models. The speed of computation is critical for large-scale economic models, and we expect applications of Julia CUDA to facilitate economic modeling and data analysis across fields.

Compliance with Ethical Standards

We don't have any conflict of interest or funding issues to report.

²¹This point is confirmed by recent work, demonstrating that agent-based simulation models with complex individual behavior can be accelerated with Thrust (see <https://flamegpu.com>).

References

- ALDRICH, E. (2014): “GPU Computing in Economics,” in *Handbook of Computational Economics*, ed. by K. Schmedders and K. Judd, North-Holland, vol. 3 of *Handbook of Computational Economics*, chap. 10, 557–598.
- ALDRICH, E., J. FERNANDEZ-VILLAVERDE, R. GALLANT, AND J. RUBIO-RAMIREZ (2011): “Tapping the supercomputer under your desk,” *Journal of Economic Dynamics and Control*.
- ARELLANO, C. (2008): “Default Risk and Income Fluctuations in Emerging Economies,” *American Economic Review*, 98, 690–712.
- ARUOBA, B. AND J. FERNANDEZ-VILLAVERDE (2015): “A comparison of programming languages,” *Journal of Economic Dynamics and Control*.
- BESARD, T. (2016): “JuliaGPU/CuArrays.jl.”
- BESARD, T., C. FOKET, AND B. DE SUTTER (2018): “Effective Extensible Programming: Unleashing Julia on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*.
- DEAN, J. AND S. GHEMAWAT (2008): “MapReduce: Simplified Data Processing on Large Clusters,” *Communications of the ACM*.
- FERNANDEZ-VILLAVERDE, J. AND D. ZARRUK (2018): “A practical guide to parallelization in economics,” *Working Papers: University of Pennsylvania*.
- GORDON, G. AND P. GUERRON-QUINTANA (2019): “A Quantitative Theory of Hard and Soft Sovereign Defaults,” *Unpublished manuscript, Boston College*.
- GUERRON-QUINTANA, P. (2016): “Sovereign Default on GPUs,” *Working Papers, Boston College*.
- GUERRON-QUINTANA, P., A. KHAZANOV, AND M. ZHONG (2021): “Nonlinear Factor Models,” *Working papers: Boston College*.
- HATCHONDO, J., L. MARTINEZ, AND H. SAPRIZA (2010): “Quantitative properties of sovereign default models: Solution methods matter,” *Review of Economic Dynamics*, 13, 919–933.
- INNES, M. J. (2017): “Generic GPU Kernels.”
- (2020): “CUDA C++ Best Practices Guide.”

JOHNSON, S. G. (2017): “More Dots: Syntactic Loop Fusion in Julia.”

KHAZANOV, A. (2021): “Sovereign Default Risk and Currency Returns,” *Unpublished manuscript Boston College*.

KIRK, D. AND W. HWU (2013): *Programming Massively Parallel Processors, 2 ed.*, Morgan Kaufmann, Waltham MA.

LETENDRE, J. (2013): “Understanding and modeling the synchronization cost in the GPU architecture.”

TAUCHEN, G. (1986): “Finite state Markov-chain approximations to univariate and vector autoregressions,” *Economics Letters*, 20, 177–181.

Appendix: Not for publication

Sovereign Default Model Implementation Code

The three major calculation components – expected value-of-default, expected value-of-repayment, and decision – are presented below:

1. Value-of-Default

```
1 #Calculate Value of Default
2 function def_init(sumdef,tau,Y,alpha)
3     iy = threadIdx().x
4     stride = blockDim().x
5     for i = iy:stride:length(sumdef)
6         sumdef[i] = CUDA.pow(exp((1-tau)*Y[i]),(1-alpha))/(1-alpha)
7     end
8     return
9 end
10
11 #adding expected value to sumdef
12 function def_add(matrix, P, beta, V0, Vd0, phi, Ny)
13     y = (blockIdx().x-1)*blockDim().x + threadIdx().x
14     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
15
16     if (iy <= Ny && y <= Ny)
17         matrix[iy,y] = beta* P[iy,y]* (phi* V0[y,1] + (1-phi)* Vd0[y])
18     end
19     return
20 end
```

```

21
22 #finish calculation of Value of Default
23 #note this final function is not a kernel and there is no return
    statement
24 function sumdef1(sumdef,Vd,Vd0,V0,phi,beta,P)
25     A = phi* V0[:,1]
26     A += (1-phi)* Vd0
27     A.= phi.* V0[:,1] .+ (1-phi).* Vd0
28     temp = P
29     temp .*= CUDA.transpose(A)
30     temp .*= beta
31     sumdef += reduce(+, temp, dims=2) #This gives Vd
32     Vd = sumdef
33 end

```

Code: Julia CUDA implementation of value-of-default

2. Value-of-Repayment

```

1 function vr(Nb,Ny,alpha,beta,tau,Vr,V0,Y,B,Price0,P)
2
3     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
4     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
5
6     if (ib <= Nb && iy <= Ny)
7
8         Max = -Inf
9         for b in 1:Nb
10             c = CUDA.exp(Y[iy]) + B[ib] - Price0[iy,b]*B[b]
11             if c > 0 #If consumption positive, calculate value of
return
12                 sumret = 0
13                 for y in 1:Ny
14                     sumret += V0[y,b]*P[iy,y]
15                 end
16
17                 vr = CUDA.pow(c,(1-alpha))/(1-alpha) + beta * sumret
18                 Max = CUDA.max(Max, vr)
19             end
20         end
21         Vr[iy,ib] = Max
22     end
23     return
24 end

```

Code: Julia CUDA implementation of value-of-repayment

3. Decision and Probability

```
1 #Calculate decision
2 function decide(Ny,Nb,Vd,Vr,V,decision)
3
4     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
5     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
6
7     if (ib <= Nb && iy <= Ny)
8
9         if (Vd[iy] < Vr[iy,ib])
10             V[iy,ib] = Vr[iy,ib]
11             decision[iy,ib] = 0
12         else
13             V[iy,ib] = Vd[iy]
14             decision[iy,ib] = 1
15         end
16     end
17     return
18 end
19
20 #Calculate probability matrix
21 function prob_calc(Ny,Nb,prob,P,decision)
22     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
23     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
24
25     if (ib <= Nb && iy <= Ny)
26         for y in Ny
27             prob[iy,ib] += P[iy,y]*decision[y,ib]
28         end
29     end
30     return
31 end
```

Code: Julia CUDA implementation of decision and probability

4. Update

```
1 err = maximum(abs.(V-V0))
2 PriceErr = maximum(abs.(Price-Price0))
3 VdErr = maximum(abs.(Vd-Vd0))
4 f(x,y) = delta * x + (1-delta) * y
5 Vd .= f.(Vd,Vd0)
6 Price .= f.(Price,Price0)
7 V .= f.(V,V0)
```


5. Temporary Matrix and Memory Allocation Example for Value-of-Repayment Calculation

```

1 #Calculate Cost Matrix C
2 function vr_C(Ny,Nb,Y,B,Price0,P,C)
3     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
4     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
5
6     if (ib <= Nb && iy <= Ny)
7         for b in 1:Nb
8             C[iy,ib,b] = -Price0[iy,b]*B[b] + CUDA.exp(Y[iy]) + B[ib]
9         end
10    end
11    return
12 end
13
14 #map C -> U(C)
15 function vr_C2(Ny,Nb,Vr,V0,Y,B,Price0,P,C,C2,sumret,alpha)
16     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
17     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
18
19     if (ib <= Nb && iy <= Ny)
20         for b in 1:Nb
21             if C[iy,ib,b] > 0
22                 c = C[iy,ib,b]
23                 C2[iy,ib,b] = CUDA.pow(c,(1-alpha)) / (1-alpha) + B[ib]
24                 - Price0[iy,b]*B[b] #Note CUDA.pow only support certain types, need
25                 to cast constant to Float32 instead of Float64
26             end
27         end
28     end
29
30     return
31 end
32
33 #Calculate sumret[iy,ib,b]
34 function vr_sumret(Ny,Nb,V0,P,sumret)
35     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
36     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
37
38     if (ib <= Nb && iy <= Ny)
39         for b in 1:Nb
40             sumret[iy,ib,b] = 0
41             for y in 1:Ny
42                 sumret[iy,ib,b] += P[iy,b]*V0[y,b]

```

```

40         end
41     end
42 end
43 return
44 end
45
46 #vr = U(c) + beta * sumret
47 function vr_add(C2, beta, sumret)
48     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
49     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
50
51     if (ib <= Nb && iy <= Ny)
52         for b in 1:Nb
53             vr[iy,ib,b] = C2[iy,ib,ib] + beta*sumret[iy,ib,ib]
54         end
55     end
56     return
57 end

```

Code: Julia CUDA code for temporary matrix and memory allocation demonstration

Benchmarking

Benchmark Code

```

1 Ny = 7 #grid number of endowment
2 Nb = 100 #grid number of bond
3 sec = 5 #number of seconds to do benchmark
4 test_rounds = 10 #number of iterations in the function for benchmarking
5
6 using Random, Distributions
7 using CUDA
8 using Base.Threads
9 using BenchmarkTools
10 #Initialization
11
12 #----Initialize Kernels
13 #line 7.1 Intitializing U((1-tau)iy) to each Vd[iy]
14 function def_init(sumdef,tau,Y,alpha)
15     iy = threadIdx().x
16     stride = blockDim().x
17     for i = iy:stride:length(sumdef)
18         sumdef[i] = CUDA.pow(exp((1-tau)*Y[i]),(1-alpha))/(1-alpha)

```

```

19     end
20     return
21 end
22
23 #line 7.2 adding second expected part to calcualte Vd[iy]
24 function def_add(matrix, P, beta, V0, Vd0, phi, Ny)
25     y = (blockIdx().x-1)*blockDim().x + threadIdx().x
26     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
27
28     #@cuprintln("iy=$iy,y=$y, stride1=$stride1, stride2=$stride2")
29     #Create 1 matrix and substract when an indice is calcualted, check
    if remaining matrix is
30     #@cuprintln("iy=$iy, y=$y")
31
32     if (iy <= Ny && y <= Ny)
33         matrix[iy,y] = beta* P[iy,y]* (phi* V0[y,1] + (1-phi)* Vd0[y])
34     end
35     return
36 end
37
38 #line 8 Calculate Vr, still a double loop inside, tried to flatten out
    another loop
39 function vr(Nb,Ny,alpha,beta,tau,Vr,V0,Y,B,Price0,P)
40
41     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
42     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
43
44     if (ib <= Nb && iy <= Ny)
45
46         Max = -Inf
47         for b in 1:Nb
48             c = CUDA.exp(Y[iy]) + B[ib] - Price0[iy,b]*B[b]
49             if c > 0 #If consumption positive, calculate value of
return
50                 sumret = 0
51                 for y in 1:Ny
52                     sumret += V0[y,b]*P[iy,y]
53                 end
54
55                 vr = CUDA.pow(c,(1-alpha))/(1-alpha) + beta * sumret
56                 Max = CUDA.max(Max, vr)
57             end
58         end
59         Vr[iy,ib] = Max
60     end

```

```

61     return
62 end
63
64
65 #line 9-14 debt price update
66 function Decide(Nb,Ny,Vd,Vr,V,decision,decision0,prob,P,Price,rstar)
67
68     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
69     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
70
71     if (ib <= Nb && iy <= Ny)
72
73         if (Vd[iy] < Vr[iy,ib])
74             V[iy,ib] = Vr[iy,ib]
75             decision[iy,ib] = 0
76         else
77             V[iy,ib] = Vd[iy]
78             decision[iy,ib] = 1
79         end
80
81         for y in 1:Ny
82             prob[iy,ib] += P[iy,y] * decision[y,ib]
83         end
84
85         Price[iy,ib] = (1-prob[iy,ib]) / (1+rstar)
86
87     end
88     return
89 end
90
91
92 #Saxpy
93 function saxpy(X,Y,delta,Nb,Ny)
94
95     ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
96     iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
97
98     if (ib <= Nb && iy <= Ny)
99         X[iy,ib] = delta* X[iy,ib] + (1-delta)* Y[iy,ib]
100     end
101     return
102 end
103
104
105 #tauchen method for creating conditional probability matrix

```

```

106 function tauchen(rho, sigma, Ny, P)
107     #Create equally spaced pts to fill into Z
108     sigma_z = sqrt((sigma^2)/(1-rho^2))
109     Step = 10*sigma_z/(Ny-1)
110     Z = -5*sigma_z:Step:5*sigma_z
111
112     #Fill in entries of 1~ny, ny*(ny-1)~ny^2
113     for z in 1:Ny
114         P[z,1] = cdf(Normal(), (Z[1]-rho*Z[z] + Step/2)/sigma)
115         P[z,Ny] = 1 - cdf(Normal(),(Z[Ny] - rho*Z[z] - Step/2)/sigma)
116     end
117
118     #Fill in the middle part
119     for z in 1:Ny
120         for iz in 2:(Ny-1)
121             P[z,iz] = cdf(Normal(), (Z[iz]-rho*Z[z]+Step/2)/sigma) -
cdf(Normal(), (Z[iz]-rho*Z[z]-Step/2)/sigma)
122         end
123     end
124 end
125
126 #Setting parameters
127
128
129 maxInd = Ny * Nb #total grid points
130 rstar = 0.017 #r* used in price calculation
131 alpha = 0.5 #alpha used in utility function
132
133 #lower bound and upper bound for bond initialization
134 lbd = -1
135 ubd = 0
136
137 #beta,phi,tau used as in part 4 of original paper
138 beta = 0.953
139 phi = 0.282
140 tau = 0.5
141
142 delta = 0.8 #weighting average of new and old matrixs
143
144 #rho,sigma For tauchen method
145 rho = 0.9
146 sigma = 0.025
147
148
149 #Initializing Bond matrix

```

```

150 #B = zeros(Nb)
151 #B = CuArray{Float32}(undef,Nb)
152 minB = lbd
153 maxB = ubd
154 step = (maxB-minB) / (Nb-1)
155 B = CuArray(minB:step:maxB) #Bond
156
157 #Intitilizing Endowment matrix
158 #Y = zeros(Ny)
159 sigma_z = sqrt((sigma^2)/(1-rho^2))
160 Step = 10*sigma_z/(Ny-1)
161 Y = CuArray(-5*sigma_z:Step:5*sigma_z) #Endowment
162
163 Pcpu = zeros(Ny,Ny) #Conditional probability matrix
164 V = CUDA.fill(1/((1-beta)*(1-alpha)),Ny, Nb) #Value
165 Price = CUDA.fill(1/(1+rstar),Ny, Nb) #Debt price
166 Vr = CUDA.zeros(Ny, Nb) #Value of good standing
167 Vd = CUDA.zeros(Ny) #Value of default
168 decision = CUDA.ones(Ny,Nb) #Decision matrix
169
170
171 U(x) = x^(1-alpha) / (1-alpha) #Utility function
172
173 #Initialize Conditional Probability matrix
174 tauchen(rho, sigma, Ny, Pcpu)
175 P = CUDA.zeros(Ny,Ny)
176 #P = CUDA.CUarray(Pcpu)
177 copyto!(P,Pcpu) #Takes long time
178
179 time_vd = 0
180 time_vr = 0
181 time_decide = 0
182 time_update = 0
183 time_init = 0
184
185 V0 = CUDA.deepcopy(V)
186 Vd0 = CUDA.deepcopy(Vd)
187 Price0 = CUDA.deepcopy(Price)
188 prob = CUDA.zeros(Ny,Nb)
189 decision = CUDA.ones(Ny,Nb)
190 decision0 = CUDA.deepcopy(decision)
191 threadcount = (32,32)
192 blockcount = (ceil(Int,Ny/32),ceil(Int,Ny/32))
193
194

```

```

195 #----Test starts
196
197 #Matrix to store benchmark results
198 Times = zeros(4)
199
200 function GPU_VD()
201     for i in 1:test_rounds
202         sumdef = CUDA.zeros(Ny)
203         @cuda threads=32 def_init(sumdef,tau,Y,alpha)
204
205         temp = CUDA.zeros(Ny,Ny)
206
207         @cuda threads=threadcount blocks=blockcount def_add(temp, P,
208             beta, V0, Vd0, phi, Ny)
209
210         temp = sum(temp,dims=2)
211         sumdef = sumdef + temp
212         for i in 1:length(Vd)
213             Vd[i] = sumdef[i]
214         end
215     end
216 end
217
218 t_vd = @benchmark GPU_VD()
219 Times[1] = median(t_vd).time/1e9/test_rounds
220 println("VD Finished")
221
222 function GPU_Decide()
223     for i in 1:test_rounds
224         @cuda threads=threadcount blocks=blockcount Decide(Nb,Ny,Vd,Vr,
225             V,decision,decision0,prob,P,Price,rstar)
226     end
227 end
228
229 t_decide = @benchmark GPU_Decide()
230 Times[3] = median(t_decide).time/1e9/test_rounds
231 println("Decide Finished")
232
233 function GPU_Update()
234     for i in 1:test_rounds
235         err = maximum(abs.(V-V0)) #These are the main time consuming
236         parts
237         PriceErr = maximum(abs.(Price-Price0))
238         VdErr = maximum(abs.(Vd-Vd0))

```

```

237
238         @cuda threads=threadcount blocks=blockcount saxpy(Vd,Vd0,delta
,1,Ny)
239         @cuda threads=threadcount blocks=blockcount saxpy(Price,Price0,
delta,Nb,Ny)
240         @cuda threads=threadcount blocks=blockcount saxpy(V,V0,delta,Nb
,Ny)
241     end
242 end
243
244 t_update = @benchmark GPU_Update()
245 Times[4] = median(t_update).time/1e9/test_rounds
246 println("Update Half Finished")
247 println("Nb=",Nb,"Ny=",Ny)
248 println(Times)
249
250
251 function GPU_VR()
252     for i in 1:test_rounds
253         @cuda threads=threadcount blocks=blockcount vr(Nb,Ny,alpha,beta
,tau,Vr,V0,Y,B,Price0,P)
254     end
255 end
256
257 t_vr = @benchmark GPU_VR() seconds = sec
258 Times[2] = median(t_vr).time/1e9/test_rounds
259
260 println("VR Finished")
261 print(dump(t_vr))
262 println("Update Fully Finished")
263 println("Nb=",Nb,"Ny=",Ny)
264 println(Times)

```

Code: Julia CUDA benchmark code

Benchmark Performance

Julia vs Julia CUDA performance sheets are presented below:

	Debt points	Vd (constant)	Vr	Decide	Update
Julia	100	1.67E-05	0.001082981	1.3362E-05	0.000035102
	175	/	0.321565953	0.002195104	2.40286E-05
	250	/	0.650970059	0.003187973	2.60898E-05
	375	/	1.31679234	0.0040663	3.11757E-05
	500	/	2.305363916	0.005614731	0.0032932
	750	/	5.36679004	0.008266143	0.00371494
	1000	/	10.3433655	0.012104929	0.00130831
	1250	/	/	0.014702145	0.00198855
	1500	/	/	0.018612573	0.00309957
	2500	/	/	0.030095512	0.00857451
	3750	/	/	0.04126148	0.00794312
	5000	/	/	0.049021429	0.00966725
	7500	/	/	0.076560898	0.01565692
	10000	/	/	0.100674108	0.02165904
	11250	/	/	0.10624682	0.00996565
	12500	/	/	0.12888245	0.0130683
	13750	/	/	0.133428585	0.00221118
	15000	/	/	0.140705525	0.00174359
	16250	/	/	0.1613247	0.001792605
	17500	/	/	0.18306948	0.00200161
	18750	/	/	0.193442	0.003633235
	20000	/	/	0.19337688	0.00411458
	Debt points	Vd (constant)	Vr	Decide	Update
Julia CUDA	100	0.00005137	0.001481135	0.00001671	0.00075366
	250	/	0.00307305	1.56999E-05	0.00093205
	500	/	0.00767661	0.00001579	0.00097521
	750	/	0.01085435	1.59199E-05	0.00099218
	1000	/	0.02962163	0.00001713	0.00093104
	1500	/	0.04524693	1.87301E-05	0.00102103
	2500	/	0.116920535	1.596E-05	0.000934375
	3750	/	0.2339817	1.54999E-05	0.00098316
	5000	/	0.46888795	0.00001615	0.000994345
	7500	/	0.89888986	1.64701E-05	0.00099719
	10000	/	1.66428956	1.62401E-05	0.00100631
	11250	/	2.080236305	0.00001582	0.00098191
	12500	/	2.71625348	1.61399E-05	0.00099651
	13750	/	3.186641435	0.00001616	0.000971585
	15000	/	3.68870817	0.00001639	0.00106833
	16250	/	4.314977035	0.0000157	0.00098324
	17500	/	5.185501565	1.61701E-05	0.00101239
	18750	/	5.856631085	0.00001588	0.00099069
	20000	/	6.526113985	0.00001696	0.0010257
	Debt points	Vd (constant)	Vr	Decide	Update
Stdpar C++	100	2.2891E-07	0.000254549	1.19654E-05	8.454E-07
	175	/	0.000733832	1.93947E-05	1.42363E-06
	250	/	0.00146435	2.80442E-05	2.02239E-06
	375	/	0.00337408	4.26068E-05	2.95258E-06
	500	/	0.00558898	4.96262E-05	3.50963E-06
	750	/	0.0119738	7.30697E-05	5.19364E-06
	1000	/	0.0211071	9.79587E-05	7.05096E-06
	1250	/	0.0359748	0.000121898	9.08545E-06
	1500	/	0.0475067	0.000153834	1.12757E-05
	2500	/	0.138043	0.000270114	1.91879E-05
	3750	/	0.33734	0.000523003	3.24313E-05
	5000	/	0.613225	0.000544792	3.51221E-05
	7500	/	1.32263	0.000790248	5.30212E-05
	10000	/	2.32161	0.00103786	7.19241E-05
	11250	/	2.9288	0.00119696	8.01766E-05
	12500	/	3.43521	0.00136703	9.60822E-05
	15000	/	4.58009	0.00158234	0.000113755
	16250	/	5.80521	0.0017704	0.000131695
	17500	/	7.69967	0.00221558	0.000148656
	18750	/	8.00612	0.00204476	0.0001499
	20000	/	9.38663	0.0022202	0.000165805

Figure 8: Julia vs Julia CUDA, Grid size $7 \times n$

	Debt points	Vd	Vr	Decide	Update
Julia	25	0.0001364	0.055270205	0.00261786	0.00000589
	50	0.00053109	0.81490575	0.020875015	5.58999E-05
	75	0.00123273	4.26258664	0.07304889	0.00005659
	100	0.00237229	13.87689287	0.16992249	0.00023132
	125	0.00357316	31.64415038	0.334118665	0.0001908
	150	0.0057178	/	0.58950077	0.00027773
	175	0.00729225	/	0.88454872	0.00036504
	200	0.009179125	/	1.33527905	0.00063912
	250	0.01428472	/	/	0.0007833
	300	0.02010567	/	/	0.0011958
	350	0.03180941	/	/	0.00189524
	400	0.0370611	/	/	0.003266055
	450	0.04510648	/	/	0.00387568
	500	0.06104513	/	/	0.005459875
	550	0.067825935	/	/	0.00658248
	600	0.08898902	/	/	0.0077394
	650	0.10190483	/	/	0.00914787
	700	0.11908736	/	/	0.01107864
	750	0.13222472	/	/	0.01116823
	Debt points	Vd	Vr	Decide	Update
Julia CUDA	100	0.0048269	0.021742605	0.00001742	0.00094093
	125	0.0060002	0.03420152	0.00002058	0.00094859
	150	0.007170599	0.04678444	2.305E-05	0.00099188
	200	0.0093563	0.16335137	0.00149893	0.00103058
	250	0.0117676	0.31863086	0.003058925	0.00104792
	300	0.014054299	0.62664573	0.00465002	0.00107269
	350	0.01640905	1.21444019	0.0092637	0.001128595
	400	0.019088199	1.96703706	0.01234142	0.00123933
	450	0.023234099	2.954906045	0.018653475	0.001266395
	500	0.022988699	4.42985646	0.02483498	0.001342405
	550	0.02863265	6.857692565	0.03282266	0.001372035
	600	0.02893155	9.17047126	0.0432739	0.001390595
	650	0.0311247	10.65110112	0.05611574	0.00143
	700	0.033381101	17.1531411	0.07028079	0.00154621
	750	0.03514595	22.27869476	0.0843548	0.0016081
	Debt points	Vd	Vr	Decide	Update
Stdpar C++	50	2.69016E-06	0.00357248	0.000222083	2.48362E-06
	75	5.27081E-06	0.0181688	0.000787701	5.64755E-06
	100	9.16423E-06	0.0609958	0.00183217	9.84192E-06
	125	1.50611E-05	0.167618	0.00429186	1.66718E-05
	150	2.22364E-05	0.350442	0.00633649	2.37501E-05
	175	0.00010776	0.706286	0.0102133	3.17238E-05
	200	3.80147E-05	1.18726	0.0149371	4.07494E-05
	250	0.000206036	3.11978	0.030141	6.90538E-05
	300	9.69854E-05	6.3988	0.0521419	0.000101558
	350	0.000181729	12.2222	0.0817523	0.000126547
	400	0.000208585	20.5257	0.125311	0.000172836
	450	0.000261079	/	0.178184	0.000224293
	500	0.000400222	/	0.246801	0.000306878
	550	0.000572342	/	0.327979	0.000398921
	600	0.00046453	/	0.450835	0.000541391
	650	0.000692935	/	0.535094	0.000635494
	700	0.000725792	/	0.679731	0.000819117
	750	0.000852845	/	0.957906	0.000971163

Figure 9: Julia vs Julia CUDA, Grid size $n \times n$

Actual Runtime

Finishing the benchmark comparison of individual components, we move to analyze the actual execution time of solving a sovereign default model with algorithm 1 in the following programming languages: Julia, C++, Julia CUDA, and C++ Standard Parallel Library. Figure 10 shows the table of execution runtimes.

The number of endowment points is fixed at 7, and the size of debt points ranges from

100 to 10,000. When the number of grid points is 100, we observe that Julia is almost as fast as C++. However, Julia becomes a faster alternative as the number of grid points increases. As a reference point, Hatchondo et al. (2010) solve a related model with 200 grid points. Their implementation takes 31 seconds. Our standard Julia implementation takes less than 10 seconds.

Julia CUDA dominates the benchmarked implementations. At $7 \times 10,000$ points, Julia CUDA executes the program at 10 times the speed of standard Julia. A huge cost comes from overhead and memory allocations, as the computation speedup of Julia CUDA compared to Julia in the main model components exceeds 1,000 times.

	Debt points	Runtime(seconds)
Julia	100	1.263
	500	10.842
	1000	36.891
	5000	830.837
	10000	3611.956
C++	100	1.211
	500	25.53
	1000	160.39
	5000	2454.5
	10000	9548.15
Julia CUDA	100	1.659
	500	3.638
	1000	6.541
	5000	83.737
	10000	347.094
C++ Stdpar	100	1.3149
	500	2.8764
	1000	7.9459
	5000	159.754
	10000	645.409

Figure 10: Speed comparison

Failed Benchmarks

In this appendix section, we present some benchmark methods that did not work in Julia CUDA. It may be very tempting to time the model directly by fitting `@timed` around each component in the implementation:

```
1 while (err > tol) & (iter < maxIter)
```

```

2   Code for Initialization
3
4   t = @timed begin
5       Code for Value-of-Default
6   end
7   time_vd += t[2]
8
9   t = @timed begin
10      Code for Value-of-Repayment
11  end
12  time_vr += t[2]
13
14  t = @timed begin
15      Code for Decision
16  end
17  time_decide += t[2]
18
19  Code for Update
20 end
21
22 time_vd /= iter
23 time_vr /= iter
24 time_decide /= iter

```

Code: failed benchmarks

This benchmark does not produce the intended result. Value-of-repayment, which should take the longest time, often produces a surprisingly shorter runtime close to zero.

Next, we turn to measuring components of the algorithm individually. We use the timing of value-of-repayment as an example.

For an individual component, time a single round of calculation:

```

1 t = @timed begin
2     @cuda threads=threadcount blocks=blockcount vr(Nb,Ny,alpha,beta,tau
3     ,Vr,V0,Y,B,Price0,P)
4 end
5 time_vr += t[2]

```

For an individual component, directly benchmark the kernel:

```

1 @benchmark @cuda threads=threadcount blocks=blockcount vr(Nb,Ny,alpha,
2     beta,tau,Vr,V0,Y,B,Price0,P)

```

For an individual component, time a single round multiple times and then take the average:

```

1 for i in 1:rounds
2     t = @timed begin
3         @cuda threads=threadcount blocks=blockcount vr(Nb,Ny,alpha,beta
4         ,tau,Vr,V0,Y,B,Price0,P)
5         ends
6         time_vr += t[2]
7     end
8 time_vr /= rounds

```

Code: Julia CPU benchmarks that fail in the Julia CUDA environment

The methods presented in this section all work for standard CPU computation but fail with Julia CUDA.