Parallel Computation of Sovereign Default Models*

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

August 26, 2021

Abstract

This paper discusses the parallel and efficient computation of macroeconomic models, with 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 GPU but bypass the steep learning and implementation costs of languages like C++ 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 detailed analysis of parallel computing, Julia-style acceleration techniques and coding recommendations. The Julia CUDA benchmark shows a substantial speedup over 1,000 times compared to standard Julia which runs on CPU. Our Julia CUDA's implementation is twice as fast as the C++ Standard Parallel Library one. We provide an accompanying Github repository with the codes and the benchmarks.

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 the framework due to its desirable programming and computation features.[1] While the

*We thank Christian Velasquez 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

[1] `https://julialang.org/`. As of June 2021, Julia 1.6 is the stable release.

paper focuses on the sovereign default model of Arellano (2008), the coding designs are relevant to researchers who want to solve complex dynamic stochastic general equilibrium models using parallelization.

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 represents many 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 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 competitive advantage through testing multiple hardware platforms and languages. The sovereign default model is solved using C++, standard parallel C++, Julia, and Julia with CUDA. The spectrum of implementations provides a comprehensive comparison on the advantages and limits of each approach. Julia and Julia CUDA stands out from the comparison due to their excellent trade-off between quick execution speed high performance and low programming barrier. The standard Julia code runs as fast as the C++ code. To give a brief picture of the effect of Julia GPU, compared to the standard Julia implementation, the speed improves by approximately 1,000 times.

Second, we provide an implementation of sovereign default model based on Julia and Julia 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.[2] Julia demonstrates an exceptional balance in execution speed and easiness of development for macroeconomics models. The choice of programming language and GPU compiler reflects our preference (another popular and highly recommended platform is Pytorch with Python). The CUDA in Julia is an actively developing library, albeit with incomplete implementation compared to CUDA in C (Besard et al., 2018).[3] The high-level Julia language and the Julia-style design of CUDA library in Julia provide an efficient process

[2]For different language implementations, check the companion Github site: `https://github.com/dengmz/ParallelDefault`

[3]The latest stable version is 3.2.1.

to test models on standard Julia and then parallelize in Julia CUDA.[4]

Due to its high learning cost, 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.[5] More recently, (Guerron-Quintana et al., 2021) demonstrate that the C++/Thrust infrastructure can be used to speedup 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 10 times speed up relative to 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 in Julia library. 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 some coding experience with popular coding languages from the readers). Section 3 revisits the sovereign default model and the solution algorithm. Section 4 walks through the coding of the solution in Julia CUDA with 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 in section 6.

[4] For 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.

[5] Thrust 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.

2 Operations and Syntax in Julia CUDA

This section introduces the Julia CUDA toolbox required to solve the sovereign default model. This paper is intended for economists who want to rapidly code and test economic theories. It does not seek to provide a comprehensive lesson on coding intricacies. We give a quick overview of the CUDA universe, but we skip over the low-level CUDA details. We recommend these in-depth and thorough internet tutorials to interested readers.[6] 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 a part of the programming platform of JuliaGPU, which aims at unifying 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 writing are likely the most compelling arguments for switching from the more mature C++ CUDA ecosystem.

Julia CUDA is the best-supported GPU platform among the JuliaGPU platforms (Innes (2020)). Julia CUDA 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 to use NVIDIA GPUs in Julia. The 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, write powerful abstraction that does not require any GPU programming experience;

2. By writing Julia CUDA kernels, compute with speed on par with kernels written in CUDA C;

3. By interfacing with CUDA APIs and libraries directly, write with the same level

[6] 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

of flexibility you would expect from a C-based programming environment (Besard (2016)).

This section focuses on the first two levels of abstractions. We will start with high-level commands like Loop Fusion and MapReduce. The CuArray and CUDA kernels are next introduced, bringing us to the second level of abstraction. Kernel programming, in particular, offers for 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 par with the well-known CUBLAS package for linear algebra operations in C++ CUDA. Loop fusion, rather than creating typical vectorized loops, gives a faster calculation with no overhead. When working with massive arrays and big data, this benefit becomes essential. The overhead in such a scenario could be in the size of gigabytes, and execution could fail due to memory shortage. For instance, on a million-element matrix, loop fusion offers performance about $4-5\times$ 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 down to seconds.

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

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

or equivalently

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

Multiple vectorized operations, like `sqrt(X)` and `*`, are fused into a single loop with no extra space requirement for temporary arrays. Among scientific programming languages, Julia is unique by offering the loop fusion feature. Other popular languages like 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 for economist not for saving the time idly waiting for the program to finish running, but by reducing the time of active coding. As will be demonstrated in the Value of Default section, loop fusion is a powerful complement to kernel programming. Kernel programming requires additional time to design and test the kernels. But before that, the researcher could quickly write loop fusion code that resembles mathematical formulas in the algorithm just by adding a few dots to the original mathematical formulas.

2.3 MapReduce

MapReduce is a widely used method to process large data sets. Google introduced MapReduce in "MapReduce: Simplified Data Processing on Large Clusters" (J Dean, 2008). Providing a simple interface, MapReduce relieves the burden of programming the detailed instructions of parallelization and optimization. In Julia CUDA, commands `map` and `reduce` enable the high-order generic array operations. Highly extensible, Julia CUDA's MapReduce can be applied to all types of arrays, including the standard Julia CUDA array type, CuArray (Besard et al., 2018). For high-level instructions, we advocate handling matrices stored in GPU by MapReduce. Such practice allows automatic parallelization of the operations for improved performance.

For example, operations `sum` and `max` are common in sovereign default models, and their implementation plays an important role in optimizing the solution algorithm. To calculate the expected value $\mathbf{E}[f(y)|y_{-}]$, one implements

$$\mathbf{E}[f(y)|y_{-}] = \sum_{\texttt{each possible value of y}} f(y) * Prob(y|y_{-})$$

for each element in the matrix $y \times y_{-}$, a combination of `reduce` and `sum` operates on the matrix and produce the result within a single line:

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

Here, P is the probability matrix $Prob(y|y_{-})$. Julia uses column-major ordering, so parameter `dims=2` indicates reducing by row.

2.4 CuArray

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

```
# generate some data on the CPU
cpu = rand(Float32, 1024)

# allocate on the GPU
gpu = CuArray{Float32}(undef,1024)

# copy from the CPU to the GPU
copyto!(gpu, cpu)

# download and verify
@test cpu == Array(gpu)
```

Code: constructing CuArray

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

2.5 Kernels

Kernels are the Julia functions for GPU. Kernels are executed in parallel on thousands of GPU threads, as opposed to normal CPU functions, which are done on a few CPU threads.

The amazing speed is one of the main reasons to program in Julia CUDA using kernels. 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 (figure 1) shows how specifically designed Julia kernels approach and exceed the performance of CUDA C kernels. Besard et al. (2018).

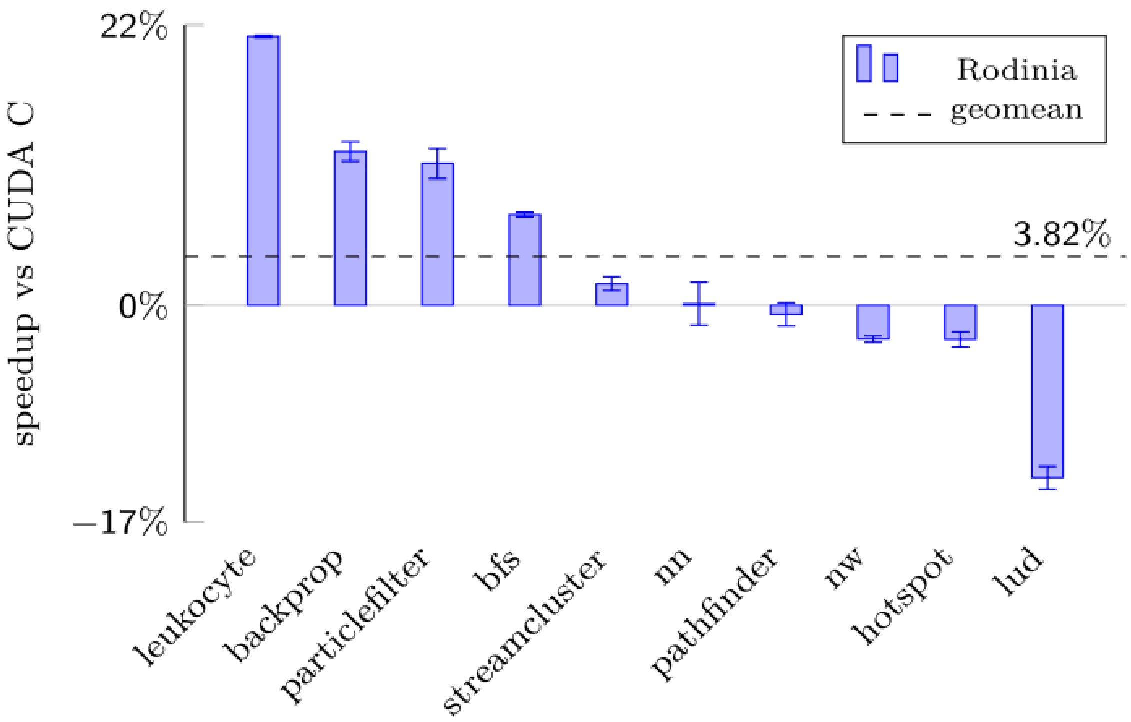

Figure 1: Relative performance of Rodinia benchmarks implemented in Julia with CUDA.jl

Designing good kernels delivers the most direct and significant performance improvements 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.

The subsection is organized into the following parts: we begin with a brief hardware-level overview of the GPU architecture, then go on to 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 to 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 2 shows the compositions of stream multiprocessors (SM) in 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 2: 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 same design follows for the grid. For example, the organization of a grid containing two-dimensional blocks of two-dimensional threads is presented in Figure 3 (Innes, 2017).

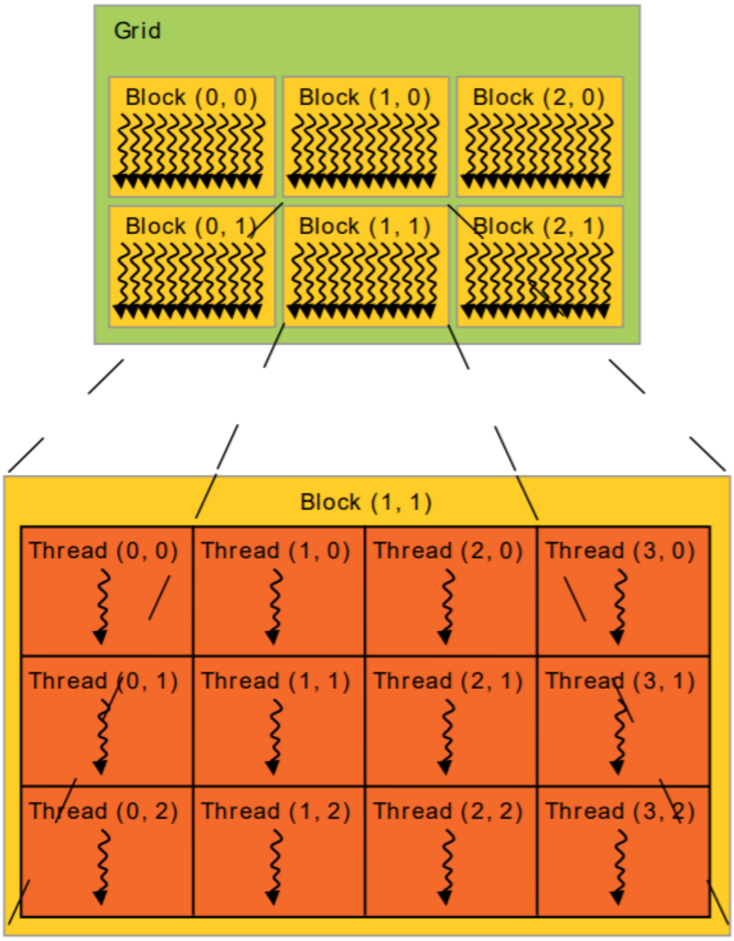

Figure 3: Grid of thread blocks

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

```
threadcount = (32,32) #two-dimensional thread blocks of size 32*32
blockcount  = (16,16) #a two-dimensional grid of size 32*32
```

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 good practice to consider hardware limits when parametrizing a thread block's size. A 32×32 thread block could be a good choice.

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 a thread within the thread block, and `blockDim` identifies the thread block containing the thread 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 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:

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

we can organize the threads into a two-dimensional matrix, with each thread uniquely assigned to a particular pair of (Nx^*, Ny^*) in the double for-loop. The kernel then calculates the output at (Nx^*, Ny^*).

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

```
x = (blockIdx().x-1)*blockDim().x + threadIdx().x
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 $[1, Nx]$ and $[1, Ny]$:

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

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

```
function example(M,Nx,Ny)
    x = (blockIdx().x-1)*blockDim().x + threadIdx().x
    y = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if ( x <= Nx && y <= Ny)
        M[x,y] += 1
    end

    return # a return statement is necessary at the end of a kernel
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 like 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.[7]

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 a natural choice of size $(32, 32)$ thread blocks. The size of the grid is the smallest grid of size over (Nx, Ny) that is covered by size $(32, 32)$ blocks. The third line demonstrates how to execute the kernel by providing the threadcount and blockcount parameters:

```
#Construct a grid of size (Nb threads) * (Ny threads)
threadcount = (32,32)
blockcount  = (ceil(Int,Nb/32),ceil(Int,Ny/32))
@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

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

Each period, the government chooses between repaying the debt obligations or defaulting payment to maximize utility of the households. 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.

[7]Kernels in Julia Cuda resemble the functor construct heavily used in C++ and CUDA programming.

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 to be 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 to whether 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 does tomorrow. But her actions tomorrow depends 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 (2016)'s parametrization:

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

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

3.1 Algorithm

The algorithm 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, the values of default and repayment are sequentially calculated if there is a feasible spending plan given the budget. A decision is made for repayment or default for each grid point, and the price of debt for the current iteration is updated.

The pseudo-code is presented below:

Algorithm 1: Psudocode Defualt Model in Standard Julia

```
Initialization;
Define maxind = N_y × N_b;
while error > tol and iter < maxiter do
    V̂ = V;
    for index = 0; index < maxind; index++ do
        Recover indices: i_b = index/N_y, i_y = index − i_b ∗ N_y;
        Compute Value-of-Default and repayment:
        V_d(i_y) = U((1 − τ)y(i_y) + βE[φV(y', 0) + (1 − φ)V_d(y') | y(i_y)];
        V_r(y(i_y), b(i_b)) = max_{b'} U(c) + βE[V(y', b') | y(i_y), b(i_b)] subject to
          constraint c ≥ 0;
        if V_r(y(i_y), b(i_b)) > V_d(y(i_y)) then
            Repay:
            V(y(i_y), b(i_b)) = V_r(y(i_y), b(i_b));
        else
            Default:
            V(y(i_y), b(i_b)) = V_d(y(i_y));
        end
        Update debt price: q(y(i_y), b(i_b)) = (1−prob(default))/(1+r*);
    end
    error = max((V̂, V));
    update Value matrix: V = δV + (1 − δ)V̂;
    iter++;
end
```

4 Sovereign in Julia

The 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 Guerron's implementation through Thrust in C CUDA. We hope to demonstrate how the Julia environment helps economists code effectively and simply.

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. The tremendous performance gain based on the simple Julia CUDA model built in this section will be demonstrated in the following benchmark section.

4.1 Value-of-Default

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

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

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

```
temp_vd = CUDA.zeros(Ny,Ny)
#Initialize
for y in 1:Ny
    for iy in 1:Ny
        temp_vd[y,iy] = f(y)*P[y,iy]
    end
end
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 speed up 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 $f(y)$:

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

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

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

Under careful design, we also can split the operations into individual components to provide further speed up:

```
A .= phi* V0[:,1]
A .+= (1-phi)* Vd0
A.= phi.* V0[:,1] .+ (1-phi).* Vd0
temp = P
temp .*= CUDA.transpose(A)
temp .*= beta
```

Code: splitted loop fusion

4.2 Value-of-Repayment

Value-of-repayment consumes the largest bulk of computation power and should be the first priority of optimization. Indeed, our improvement of the value-of-repayment calculation makes the greatest contribution to the total performance speedup. The expensive computation cost comes from an expected value calculation with four variables of computation complexity $O(n^4)$.[8] In our CPU design, Julia-style linear algebra and loop fusion operations again replace the standardized for-loops.

```
for ib in 1:Nb
    for iy = 1:Ny
        Max = -Inf
```

[8]Since 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.

```
        for b in 1:Nb
            c = exp(Y[iy]) + B[ib] - Price0[iy,b]*B[b]
            if c > 0
                sumret = 0
                for y in 1:Ny
                    sumret += P[iy,y]*V0[y,b]
                end
                vr = U(c) + beta * sumret
                Max = max(Max, vr)
            end
        end
    Vr[iy,ib] = Max
    end
end
```

Code: Julia CPU code for the Value-of-Repayment algorithm

In Guerron-Quintana (2016)'s 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: large variation of execution time among the sizable threads may greatly impact overall performance. During every round of computation, the quicker 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 inspect the structure of the Value-of-Repayment problem to assign the dimension of the thread blocks and the grid. Recall that 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 $Ny \times Nb$ grid. Therefore, we need the last kernel to fill the value into the matrix `Vr`, which contains the value of repayment. It seems tempting to encapsulate all four loops through a two-dimensional thread block and a two-dimensional grid, thus reducing individual kernel runtime to minimal. The memory transfer and storage conditions, however, we suggest avoiding this easy design. Each thread requires the Price matrix and the original Value matrix from GPU memory: `sumret += P[iy,y]*V0[y,b]`. Suppose we parametrized an endowment×debt grid with $Ny = 500$, $Nb = 500$, and thread blocks of size $(32, 32)$. As a thread block shares the local memory, a total of about $2 \times 500^2 \times 500^2/32^2 = 192,312$ transfers of floats will be required just to begin the calculation.

Instead, we make a tradeoff of speed for efficient coding. 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 $Ny \times Nb$ grid.

```
function vr(Nb,Ny,alpha,beta,tau,Vr,V0,Y,B,Price0,P)

    ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if (ib <= Nb && iy <= Ny)

        Max = -Inf
        for b in 1:Nb
            c = CUDA.exp(Y[iy]) + B[ib] - Price0[iy,b]*B[b]
            if c > 0 #If consumption positive, calculate value of return
                sumret = 0
                for y in 1:Ny
                    sumret += V0[y,b]*P[iy,y]
                end

                vr = CUDA.pow(c,(1-alpha))/(1-alpha) + beta * sumret
                Max = CUDA.max(Max, vr)
            end
        end
        Vr[iy,ib] = Max
    end
    return
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 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 a 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. The benchmark is executed on Julia REPL in Atom.[9]

5.1 Julia CUDA

The benchmark analysis is performed on the three major computation components: Value-of-Default, Value-of-Repayment, and Decision. Figure 4 plots the results from the Julia CUDA benchmark. The X-axis depicts the size of grid points for the Endowment$\times$Bond Matrix of size $Ny \times Nb$, and the Y-axis measures the median running time in microseconds for one evaluation of each component of the model. We address the variance across simulations in subsection 5.4 to guarantee stable performance. After assuring that the variance of performance across simulations is low, the main result of the section, the median benchmark runtime is a good estimation of the actual performance.

The benchmark focuses on two parameterization with different emphasis. In this subsection we introduce the first benchmark, which aims to maximize the difference in complexity magnitude among the components. In this benchmark, the Endowment $\times$ Bond grid is a square matrix containing n possible endowment values and n possible bond values ($n = Ny = Nb$). As a result, the Value-of-Repayment component will dominate runtime, and Julia CUDA will deliver the greatest speedup in this component. The time complexity of each component is summarized as follows:

Value-of-Default	Value-of-Repayment	Decision
$O(Ny^2)$ or $O(n^2)$	$O(Ny^2 \cdot Nb^2)$ or $O(n^4)$	$O(Ny \cdot Nb)$ or $O(n^2)$

Table 1: Time Complexity of the components in Sovereign Default Model

The figure 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 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 $Ny = Nb = 500$ requires 500^3 of Float32 memory[10], equivalent to $500^3 \times 32/10^9 = 4$GB of RAM. At $Ny = Nb = 800$, the required RAM memory reaches 16 GB. Further granulation would be uneconomical on a personal computer.[11]

[9] The 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.

[10] Using Float64 on GPU is prone to error

[11] A quick fix of the memory issue is to remove the high-dimensional matrix that had speed up the

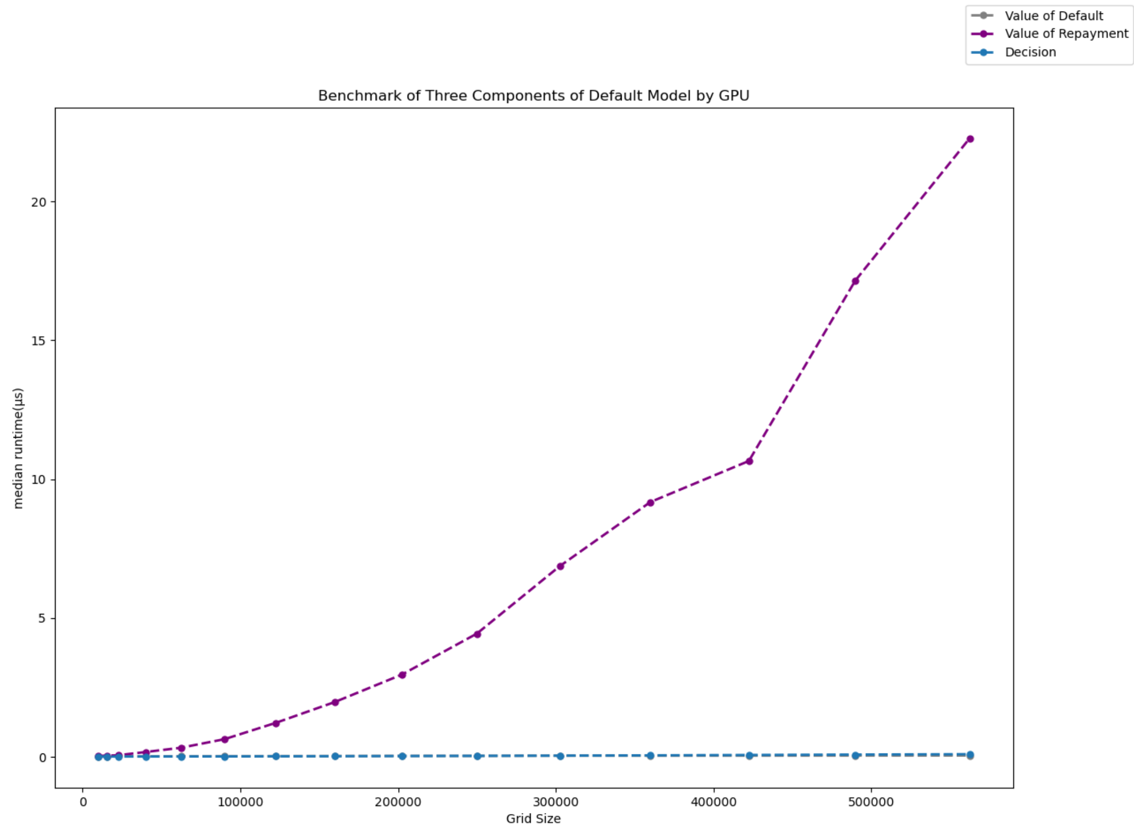

Figure 4: GPU Benchmark of sovereign default model

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 introduce the second benchmark test aimed to realistic settings. This benchmark follows a canon parametrization of 7 endowment points with varying debt granularization of $Nb = n$ debt points. With a constant Ny, the complexity is as follows:

Value-of-Default	Value-of-Repayment	Decision
$O(1)$	$O(Nb^2)$ or $O(n^2)$	$O(Nb)$ or $O(n)$

Table 2: Time Complexity of the Second Benchmark

The following graph demonstrates GPU advantage over CPU in Julia. Note the Value-of-Repayment component dominates runtime. The component has complexity $O(n^2)$. We

calculation. In the trade-off of memory for speed, the standard vectorized loops replace the MapReduce 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.

ignore the Value-of-Default part given its constant complexity.

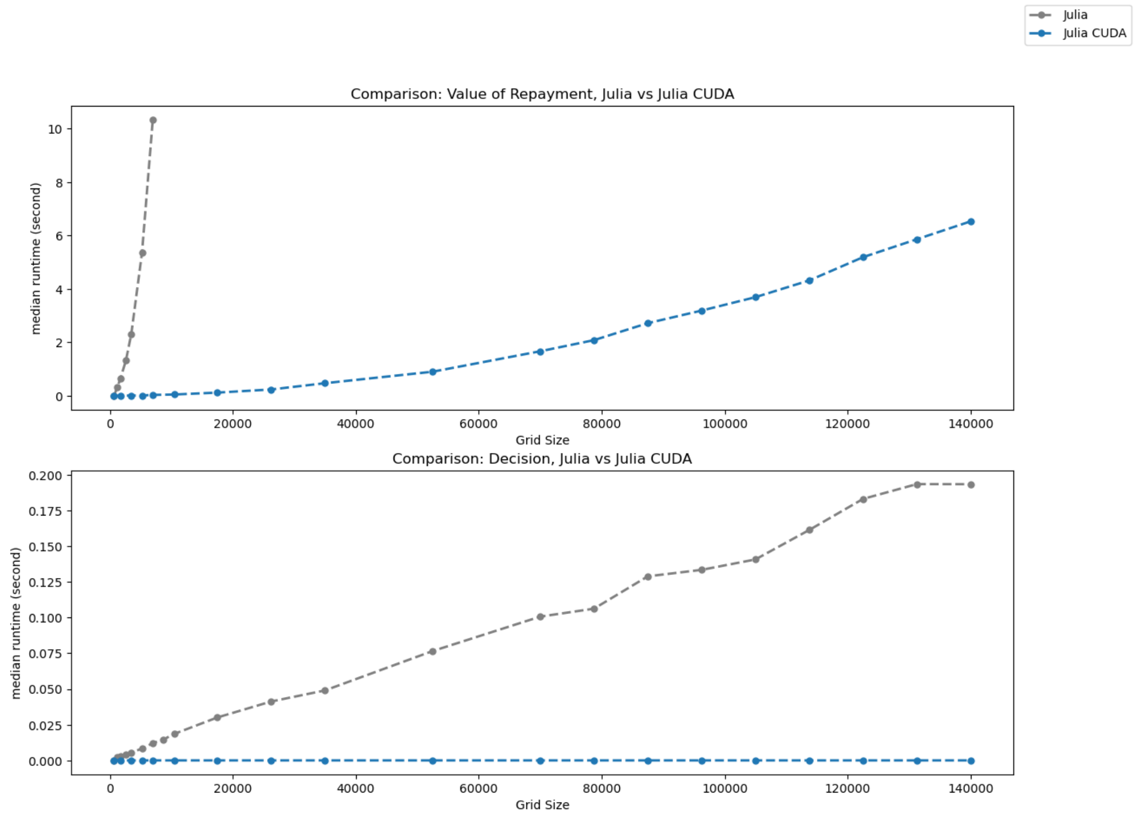

Figure 5: First Benchmark of grid size $7 \times Nb$. Comparison of the last two components in the model

The second benchmark is performed on a Endowment×Debt grid of size $Ny \times Nb$, $Ny = Nb$, and the time complexity is given in table 3.

Value-of-Default	Value-of-Repayment	Decision
$O(Ny^2)$ or $O(n^2)$	$O(Ny^2 \cdot Nb^2)$ or $O(n^4)$	$O(Ny \cdot Nb)$ or $O(n^2)$

Table 3: Time Complexity of the Second Benchmark

Figure 6 demonstrates the dominating performance of GPU given two extra complexity orders. The program's runtime performance is dominated by the most time-consuming process, Value-of-Repayment calculation. This computation has complexity of $O(Ny^2 Nb^2) = O(n^4)$ for Endowment × Bond matrix of size $n \times n$. For CPU computation, the runtime of value-of-repayment calculation grows in polynomial scale to over 10 seconds on a 100×100 size 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.[12]

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

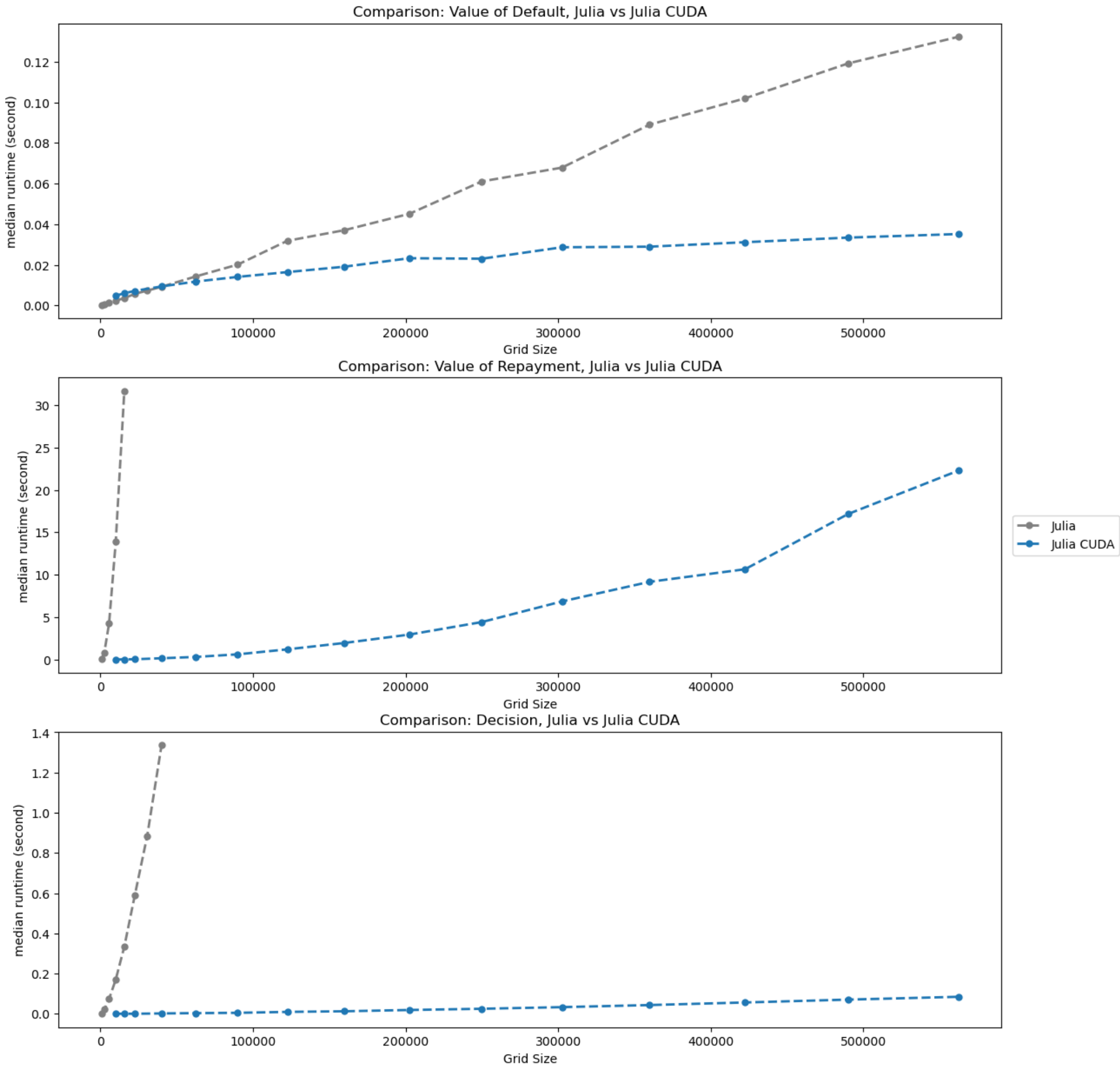

Figure 6: Second Benchmark of grid size $Ny \times Nb, Ny = Nb$. Comparison of the three components in the model

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 of Value-of-Repayment (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 aimed to circumvent the technical difficulties of programming CUDA with standard C++ code. Using features of object-oriented programming, all data grids and calculation components are encapsulated in a single class and then processed in C++ Stdpar code to run in CUDA. The design simplifies the code transfer process from standard C++ to C++ Stdpar.

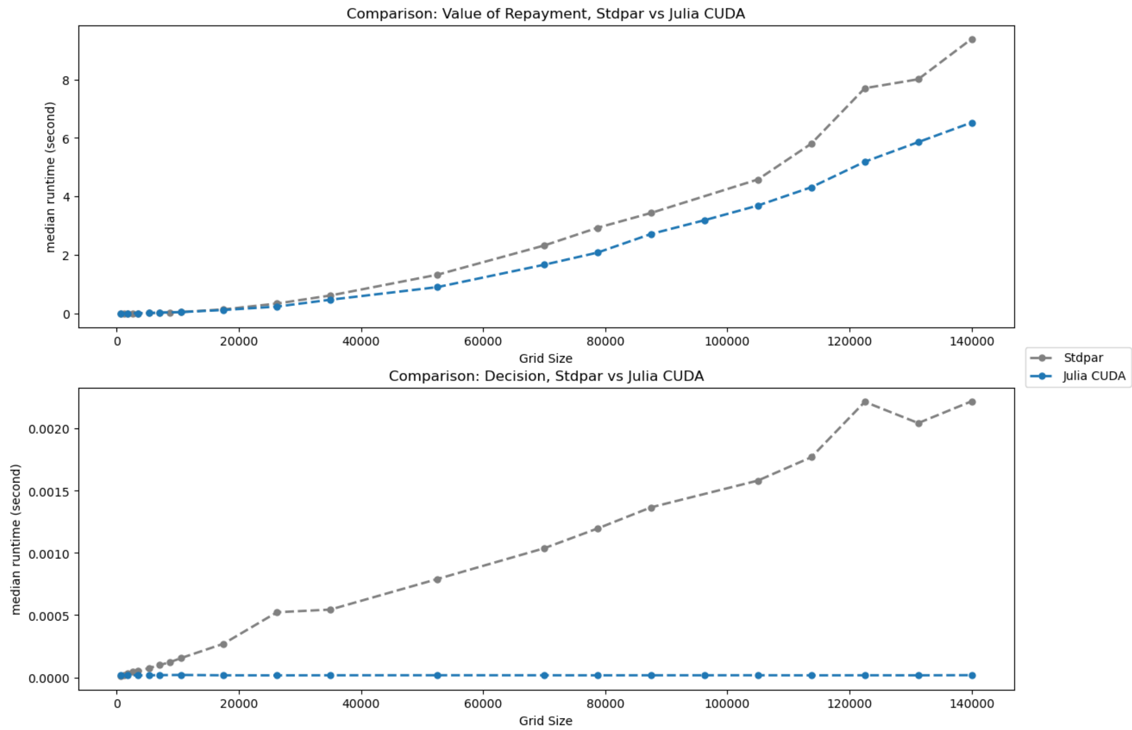

Figure 7: First Benchmark of grid size $7 \times Nb$. Comparison of the last two components in the model.

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 confirmed Julia CUDA's advantage over Stdpar by benchmarking real execution of the entire model.

5.4 Runtime Distribution

The benchmark exercise shows the satisfactory average performance of Julia CUDA running the components of the sovereign default Model. Now we'll look into the potential fluctuations of the kernels' performances in real-time. This is accomplished by sampling the distributions of run-time for the three kernels of Value-of-Default, Value-of-Repayment, and the Decision. If the same delay happens in several execution iterations, that is, when the model is solved by value function iteration, any outlier could drastically slow down performance.

We present boxplots for two sets of parameters of $(Ny = 100, Nb = 100)$ in Figure 9 and $(Ny = 7, Nb = 1000)$ in Figure 10. Each boxplot displays the distribution of run-time speed for the three kernels. The longest execution time is due to the pre-compilation feature of the Julia language: kernels take extra time for pre-compilation in the first round

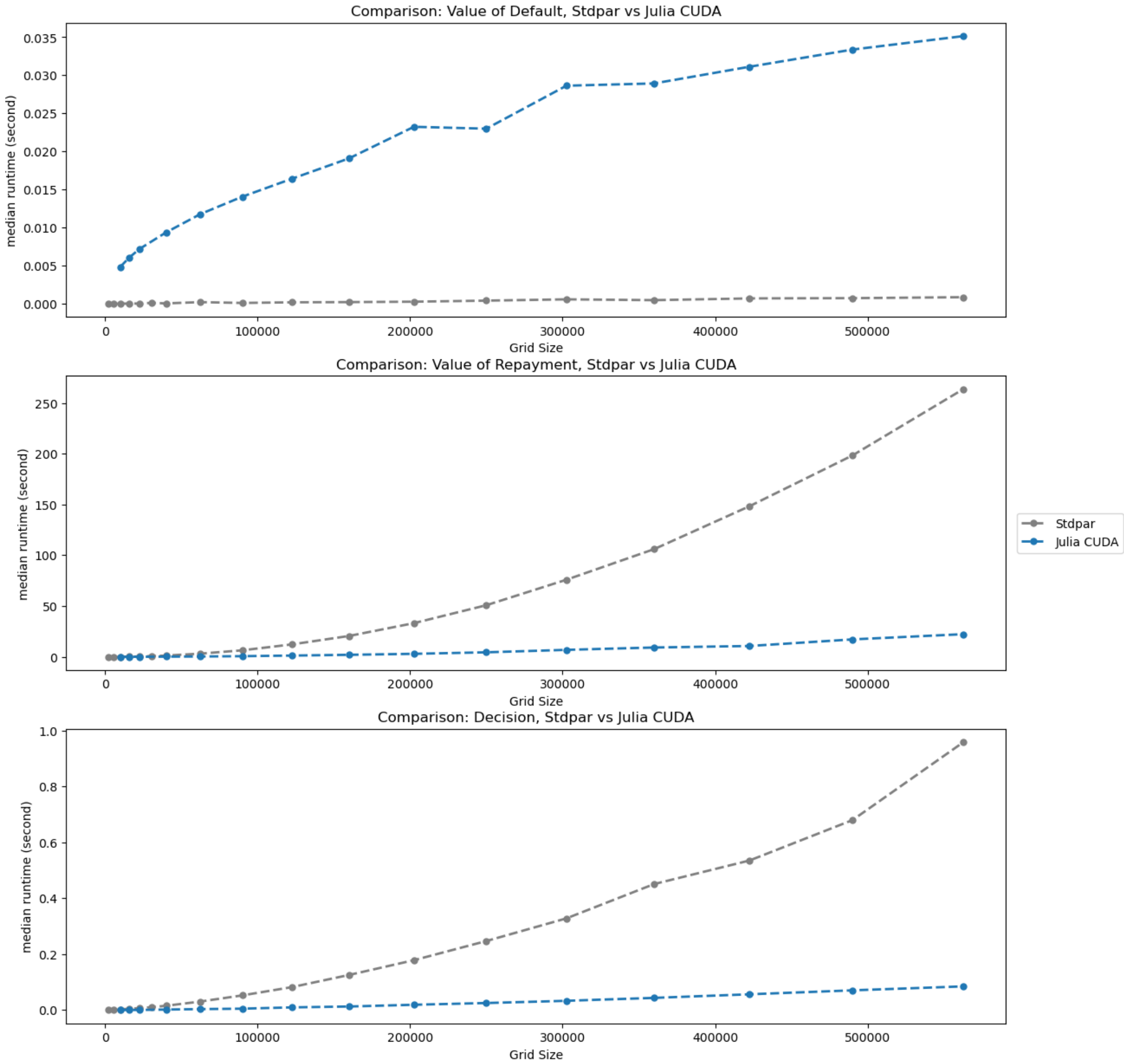

Figure 8: Second Benchmark of grid size $Ny \times Nb, Ny = Nb$. Comparison of the three components in the model

(circles in the figures). Performance of the kernels after the first iteration is shorter in run time and displays far less fluctuation. In real-time execution, the kernel will run for hundreds of iterations and the impact of the long pre-compilation in the first round will be minimized. As a result, Julia CUDA's performance in running the sovereign default model will be relatively stable in real-time.

5.5 Actual Runtime

This subsection compares the actual execution time for the solution of the sovereign default model by running algorithm 1 with the following programming languages: Julia, C++, Julia CUDA, and C++ Standard Parallel Library (Figure 11). 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++.

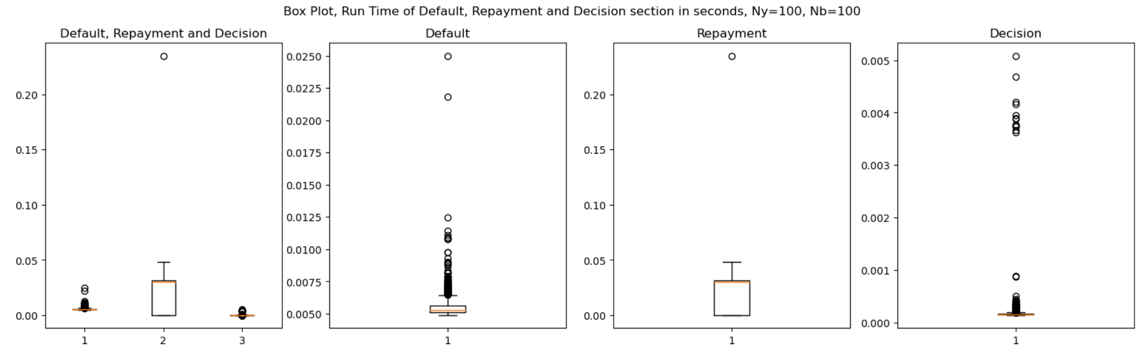

Figure 9: Runtime Distribution: Ny=100, Nb=100

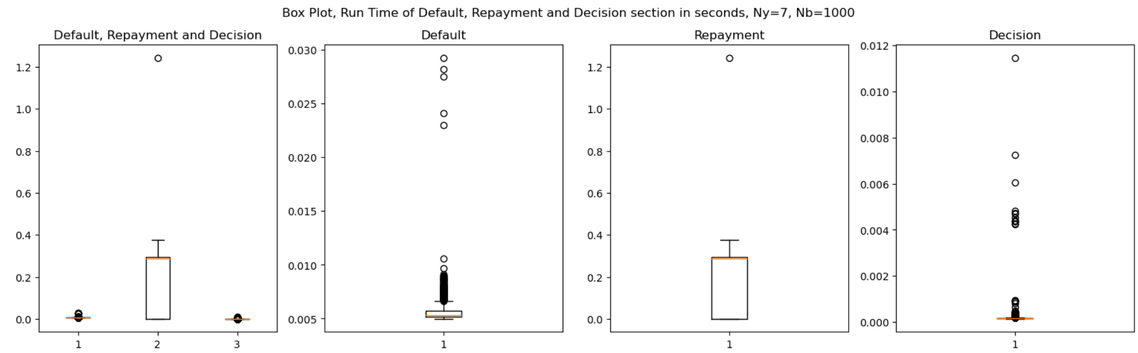

Figure 10: Runtime Distribution: Ny=7, Nb=1000

However, Julia becomes a faster alternative as the grid points increase. 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 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.

5.6 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 give a simple and feasible way to benchmark Julia CUDA with the `BenchmarkTools` library through 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 on a Julia

	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 11: Speed Comparison

CUDA kernel does not produce the correct result, possibly due to compatibility issue between standard Julia platform and the Julia CUDA package. Instead, we introduced an extra Julia function, which iterates and computes the kernel for multiple loops. 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 dispropotional time on overhead. The standard Julia benchmark follows the same design. The following code benchmarks Value-of-Repayment:

```
sec=120 #set time allowance for benchmarking
test_rounds = 10 #number of rounds to iter the kernel

function GPU_VR()
    for i in 1:test_rounds
        @cuda threads=threadcount blocks=blockcount vr(Nb,Ny,alpha,beta
    ,tau,Vr,V0,Y,B,Price0,P)
    end
end
```

```

t_vr = @benchmark GPU_VR() seconds = sec
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 launched on Aug 7, 2018, and is the oldest version still supported. The paper uses Julia 1.4.2 (2020-05-23) and the latest update is version 1.6.1 (2021-04). The ecosystem is thriving, but still relatively small. This section aims to point out some pitfalls to avoid based on the 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 the future transfer of coding design across programming platforms outweighs the cost and coding effort to dissect the algorithm.

Synchronization cost arises when various threads within the GPU need to share data among themselves (Letendre (2013)). Ideally, the work in each thread is independent, meaning each of the many threads in a GPU runs a kernel function in parallel, with no need to share information between threads. However, the implementation of most economic problems requires a more complex control flow that requires communication across threads.

The control flow of economic models often requires a thorough update of values before proceeding to the next computation phase. 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×debt point, namely to calculate the value-of-default, value-of-repayment, decision, and price values given the particular endowment×debt index. To derive the debt price at a particular endowment×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 run-time, the matrices of Value-of-

Default, Value-of-Repayment, and Decision must be fully updated by the respective three kernels before any single price point could be correctly derived.

The difference in computation time across threads and blocks indicates that some threads will finish before the others. The wait time for slower threads to finish before synchronizing the updated information is called the synchronization cost. We can label two types of synchronization costs: the maximum synchronization cost from the wait for data access, and the minimum synchronization cost from waiting for the slowest warp to finish. We recommend partitioning large kernels to improve performance.[13] Such division has multiple benefits. Compared to a single kernel function, run-time variance among kernels is drastically reduced. Also, debugging and feature-adding are much more efficient in a modular design.

However, we warn the potential costs of dividing kernels. Redundant kernel initialization and memory transfer speed could remove any intended speed improvement. Fortunately, thanks to CUDA's power, there is often no need to investigate into the details of the kernels to produce results in a satisfactory time.

6.2 Memory Management through Garbage Collection

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

Julia CUDA automatically stores user-created objects and cache the underlying memory in a memory pool to speed up future allocations. When memory pressure is high, the memory pool automatically free 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 a out-of-memory situation, as will be discussed in the next subsection on temporary matrix, reclaiming cached memory will not solve the problem.

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

Batching iterator provides an interface to input data beyond the memory capacity of GPU with automatic memory free up. An official example of using `CuIterator` to batch operations is provided below (Besard (2016)):

```
batches = [([1],[2]),([3],[4])]

```

[13] a good reference is Letendre, J.(2013): "Understanding and modeling the synchronization cost in the GPU architecture"

```
for (batch, (a,b)) in enumerate(CuIterator(batches))
    println("Batch $batch: ", a .+ b)
end

Batch 1: [3]
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 GPU. The example code performs a matrix addition on the two matrices in each chunk. When loading a new batch onto the GPU, CuIterator eagerly frees the old batch from the GPU memory through command `unsafe_free!`. This automatic process removes the need of manual memory free up or Julia garbage collector in GPU. Note that while easing the GPU memory allocation pressure, 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 endowment×debt requires access to all possible states of endowment×debt, thus access to the entire endowment×debt matrix. Dividing the endowment×debt matrix into batches effectively blocks access to the entire matrix for each batch.

6.3 Temporary Matrix

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

For instance, in the appendix, we included calculation of Value-of-Repayment through temporary matrices. Several temporary matrices were created to facilitate the calculation. Take the cost matrix $C[iy, ib, b]$ for example, it is a temporary matrix containing three variables. Allocating the cost matrix beforehand on the device/GPU removes 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 a speed-up compared to 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 calculation could be handled by a kernel, with the output stored for the next step. Another benefit is that temporary matrices could utilize the built-in functions of Julia CUDA for quick matrix manipulations.

The disadvantages of temporary matrices are threefold. Due to these deficiencies, our standard implementation for Julia CUDA does not utilize temporary matrices. Firstly, temporary matrices can be large. Take the cost matrix $C[iy, ib, b]$ for example. Suppose there are 7 endowment levels with $10,000$ debt levels, the size of C reaches 5.215 GiB.

The second disadvantage makes temporary matrices sometimes 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, "898 GiB/s on the NVIDIA Tesla V100" to "16 GiB/s on the PCIe x16 Gen3." It is therefore critical to avoid storage on the host CPU. Instead, it is more preferable to run kernels on GPU to minimize data transfer to the host CPU. However, this would reintroduce the first disadvantage as the memory is stored on GPU, thus leading to a dilemma between storing the matrix on CPU or GPU.

Continue with the example of the cost matrix. Suppose the cost matrix is stored on GPU memory, this would leave to a potential overflow. 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. Suppose the matrix is instead stored on the host CPU, for each of the three hundred rounds of value iteration, the 5.215 GiB matrix must be transferred from the CPU to GPU and back again. If 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[14]. With more temporary matrices, the data transfer cost becomes highly undesirable.[15]

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 of loop fusion comes by removing the costly allocation and deallocation process in creating temporary arrays. However, automatically fusing multiple broadcast operations does not show the level of improvement as expected. The speedup of loop

[14]300 is a rough estimation of the number of iterations required to run the Sovereign Default algorithm with error tolerance 1e$-$6

[15]In 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

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, consider the following snippet to calculate $temp = \beta P(A)^T$:

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

is much faster than

```
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 similar reasons as 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.[16]

The complexity of control flows in a model could escalate the challenge in kernel designs and run-time performance. 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 of 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 state-of-art heterogeneous agent models will experience significant speedup improvements compared to serial implementations.[17]

The limit of memory transfer bandwidth, as pointed out in the temporary matrix subsection, could slowdown runtime performance. If we can not fit all data in GPU memory, we have to copy the memory from CPU to GPU, and feed the updates back

[16] Particular types of computer science problems, such as graph algorithm, searching, and sorting fall into this category. But given their relatively little importance in Economics than in Computer Science, we do not go into details about these interesting fields.

[17] This point is confirmed by recent work, demonstrating that agent-based simulation models with complex individual behavior can be accelerated with Thrust (see flamegpu)

from GPU to CPU. The runtime transfer cost between GPU and CPU could dominate the actual computation cost spent in GPU. We've introduced one method in memory management subsection to improve the situation: use BatchIterator to perform large-scale data transfer from CPU to 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. Julia's syntax, in contrast to Python/C++, adheres to several mathematical traditions, such as 1-indexing rather than 0-indexing, delivering an excellent experience for developing economic models. This paper focuses on computational details and coding standards for parallel computation in Julia. Using the sovereign 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.

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.*

Fernandez-Villaverde, J. and D. Zarruk (2018): "A practical guide to parallelization in Economics," *Working Paper: U. 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 paper Boston College.*

Guerron-Quintana, P., A. Khazanov, and M. Zhong (2021): "Nonlinear Factor Models," *Working paper: 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," .

J Dean, S. G. (2008): "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM.*

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 Parellel Processors*, Morgan Kaufmann, MA, 2 ed.

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

```
#Calculate Value of Default
function def_init(sumdef,tau,Y,alpha)
    iy = threadIdx().x
    stride = blockDim().x
    for i = iy:stride:length(sumdef)
        sumdef[i] = CUDA.pow(exp((1-tau)*Y[i]),(1-alpha))/(1-alpha)
    end
    return
end

#adding expected value to sumdef
function def_add(matrix, P, beta, V0, Vd0, phi, Ny)
    y = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if (iy <= Ny && y <= Ny)
        matrix[iy,y] = beta* P[iy,y]* (phi* V0[y,1] + (1-phi)* Vd0[y])
    end
    return
end

#finish calculation of Value of Default
#note this final function is not a kernel and there is no return
   statement
function sumdef1(sumdef,Vd,Vd0,V0,phi,beta,P)
    A = phi* V0[:,1]
    A += (1-phi)* Vd0
    A.= phi.* V0[:,1] .+ (1-phi).* Vd0
    temp = P
    temp .*= CUDA.transpose(A)
    temp .*= beta
```

```
    sumdef += reduce(+, temp, dims=2) #This gives Vd
    Vd = sumdef
end
```

Code: Julia CUDA implementation of Value-of-Default

2. Value-of-Repayment

```
function vr(Nb,Ny,alpha,beta,tau,Vr,V0,Y,B,Price0,P)

    ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if (ib <= Nb && iy <= Ny)

        Max = -Inf
        for b in 1:Nb
            c = CUDA.exp(Y[iy]) + B[ib] - Price0[iy,b]*B[b]
            if c > 0 #If consumption positive, calculate value of return
                sumret = 0
                for y in 1:Ny
                    sumret += V0[y,b]*P[iy,y]
                end

                vr = CUDA.pow(c,(1-alpha))/(1-alpha) + beta * sumret
                Max = CUDA.max(Max, vr)
            end
        end
        Vr[iy,ib] = Max
    end
    return
end
```

Code: Julia CUDA implementation of Value-of-Repayment

3. Decision and Probability

```
#Calculate decision
function decide(Ny,Nb,Vd,Vr,V,decision)

    ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if (ib <= Nb && iy <= Ny)

```

```
        if (Vd[iy] < Vr[iy,ib])
            V[iy,ib] = Vr[iy,ib]
            decision[iy,ib] = 0
        else
            V[iy,ib] = Vd[iy]
            decision[iy,ib] = 1
        end
    end
    return
end

#Calculate probability matrix
function prob_calc(Ny,Nb,prob,P,decision)
    ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if (ib <= Nb && iy <= Ny)
        for y in Ny
            prob[iy,ib] += P[iy,y]*decision[y,ib]
        end
    end
    return
end
```

Code: Julia CUDA implementation of Decision and prabability

4. Update

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

Code: Julia CUDA implementation of Update

5. Temporary Matrix and Memory Allocation example for Value-of-Repayment calculation

```
#Calculate Cost Matrix C
function vr_C(Ny,Nb,Y,B,Price0,P,C)
    ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
```

```

    if (ib <= Nb && iy <= Ny)
        for b in 1:Nb
            C[iy,ib,b] = -Price0[iy,b]*B[b] + CUDA.exp(Y[iy]) + B[ib]
        end
    end
    return
end

#map C -> U(C)
function vr_C2(Ny,Nb,Vr,V0,Y,B,Price0,P,C,C2,sumret,alpha)
    ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if (ib <= Nb && iy <= Ny)
        for b in 1:Nb
            if C[iy,ib,b] > 0
                c = C[iy,ib,b]
                C2[iy,ib,b] = CUDA.pow(c,(1-alpha)) / (1-alpha) + B[ib]
    - Price0[iy,b]*B[b] #Note CUDA.pow only support certain types, need
    to cast constant to Float32 instead of Float64
            end
        end
    end
    return
end

#Calcuate sumret[iy,ib,b]
function vr_sumret(Ny,Nb,V0,P,sumret)
    ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if (ib <= Nb && iy <= Ny)
        for b in 1:Nb
            sumret[iy,ib,b] = 0
            for y in 1:Ny
                sumret[iy,ib,b] += P[iy,b]*V0[y,b]
            end
        end
    end
    return
end

#vr = U(c) + beta * sumret
function vr_add(C2, beta, sumret)
```

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

    if (ib <= Nb && iy <= Ny)
        for b in 1:Nb
            vr[iy,ib,b] = C2[iy,ib,ib] + beta*sumret[iy,ib,ib]
        end
    end
    return
end
```

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

Benchmarking

Benchmark Code

```
Ny = 7 #grid number of endowment
Nb = 100 #grid number of bond
sec = 5 #number of seconds to do benchmark
test_rounds = 10 #number of iterations in the function for benchmarking

using Random, Distributions
using CUDA
using Base.Threads
using BenchmarkTools
#Initialization

#----Initialize Kernels
#line 7.1 Intitializing U((1-tau)iy) to each Vd[iy]
function def_init(sumdef,tau,Y,alpha)
    iy = threadIdx().x
    stride = blockDim().x
    for i = iy:stride:length(sumdef)
        sumdef[i] = CUDA.pow(exp((1-tau)*Y[i]),(1-alpha))/(1-alpha)
    end
    return
end

#line 7.2 adding second expected part to calcualte Vd[iy]
function def_add(matrix, P, beta, V0, Vd0, phi, Ny)
    y = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y
```

```julia

    #@cuprintln("iy=$iy,y=$y,stride1=$stride1,stride2=$stride2")
    #Create 1 matrix and substract when an indice is calcualted, check if remaining matrix is
    #@cuprintln("iy=$iy, y=$y")

    if (iy <= Ny && y <= Ny)
        matrix[iy,y] = beta* P[iy,y]* (phi* V0[y,1] + (1-phi)* Vd0[y])
    end
    return
end

#line 8 Calculate Vr, still a double loop inside, tried to flatten out another loop
function vr(Nb,Ny,alpha,beta,tau,Vr,V0,Y,B,Price0,P)

    ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if (ib <= Nb && iy <= Ny)

        Max = -Inf
        for b in 1:Nb
            c = CUDA.exp(Y[iy]) + B[ib] - Price0[iy,b]*B[b]
            if c > 0 #If consumption positive, calculate value of return
                sumret = 0
                for y in 1:Ny
                    sumret += V0[y,b]*P[iy,y]
                end

                vr = CUDA.pow(c,(1-alpha))/(1-alpha) + beta * sumret
                Max = CUDA.max(Max, vr)
            end
        end
        Vr[iy,ib] = Max
    end
    return
end


#line 9-14 debt price update
function Decide(Nb,Ny,Vd,Vr,V,decision,decision0,prob,P,Price,rstar)

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

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

    if (ib <= Nb && iy <= Ny)

        if (Vd[iy] < Vr[iy,ib])
            V[iy,ib] = Vr[iy,ib]
            decision[iy,ib] = 0
        else
            V[iy,ib] = Vd[iy]
            decision[iy,ib] = 1
        end

        for y in 1:Ny
            prob[iy,ib] += P[iy,y] * decision[y,ib]
        end

        Price[iy,ib] = (1-prob[iy,ib]) / (1+rstar)

    end
    return
end


#Saxpy
function saxpy(X,Y,delta,Nb,Ny)

    ib = (blockIdx().x-1)*blockDim().x + threadIdx().x
    iy = (blockIdx().y-1)*blockDim().y + threadIdx().y

    if (ib <= Nb && iy <= Ny)
        X[iy,ib] = delta* X[iy,ib] + (1-delta)* Y[iy,ib]
    end
    return
end


#tauchen method for creating conditional probability matrix
function tauchen(rho, sigma, Ny, P)
    #Create equally spaced pts to fill into Z
    sigma_z = sqrt((sigma^2)/(1-rho^2))
    Step = 10*sigma_z/(Ny-1)
    Z = -5*sigma_z:Step:5*sigma_z

    #Fill in entries of 1~ny, ny*(ny-1)~ny^2
    for z in 1:Ny
```

```
        P[z,1] = cdf(Normal(), (Z[1]-rho*Z[z] + Step/2)/sigma)
        P[z,Ny] = 1 - cdf(Normal(),(Z[Ny] - rho*Z[z] - Step/2)/sigma)
    end

    #Fill in the middle part
    for z in 1:Ny
        for iz in 2:(Ny-1)
            P[z,iz] = cdf(Normal(), (Z[iz]-rho*Z[z]+Step/2)/sigma) -
   cdf(Normal(), (Z[iz]-rho*Z[z]-Step/2)/sigma)
        end
    end
end

#Setting parameters


maxInd = Ny * Nb #total grid points
rstar = 0.017 #r* used in price calculation
alpha = 0.5 #alpha used in utility function

#lower bound and upper bound for bond initialization
lbd = -1
ubd = 0

#beta,phi,tau used as in part 4 of original paper
beta = 0.953
phi = 0.282
tau = 0.5

delta = 0.8 #weighting average of new and old matrixs

#rho,sigma For tauchen method
rho = 0.9
sigma = 0.025


#Initializing Bond matrix
#B = zeros(Nb)
#B = CuArray{Float32}(undef,Nb)
minB = lbd
maxB = ubd
step = (maxB-minB) / (Nb-1)
B = CuArray(minB:step:maxB) #Bond

#Intitializing Endowment matrix
```

```
#Y = zeros(Ny)
sigma_z = sqrt((sigma^2)/(1-rho^2))
Step = 10*sigma_z/(Ny-1)
Y = CuArray(-5*sigma_z:Step:5*sigma_z) #Endowment

Pcpu = zeros(Ny,Ny)  #Conditional probability matrix
V = CUDA.fill(1/((1-beta)*(1-alpha)),Ny, Nb) #Value
Price = CUDA.fill(1/(1+rstar),Ny, Nb) #Debt price
Vr = CUDA.zeros(Ny, Nb) #Value of good standing
Vd = CUDA.zeros(Ny) #Value of default
decision = CUDA.ones(Ny,Nb) #Decision matrix


U(x) = x^(1-alpha) / (1-alpha) #Utility function

#Initialize Conditional Probability matrix
tauchen(rho, sigma, Ny, Pcpu)
P = CUDA.zeros(Ny,Ny)
#P = CUDA.CUarray(Pcpu)
copyto!(P,Pcpu) #Takes long time

time_vd = 0
time_vr = 0
time_decide = 0
time_update = 0
time_init = 0

V0 = CUDA.deepcopy(V)
Vd0 = CUDA.deepcopy(Vd)
Price0 = CUDA.deepcopy(Price)
prob = CUDA.zeros(Ny,Nb)
decision = CUDA.ones(Ny,Nb)
decision0 = CUDA.deepcopy(decision)
threadcount = (32,32)
blockcount = (ceil(Int,Ny/32),ceil(Int,Ny/32))


#----Test starts

#Matrix to store benchmark results
Times = zeros(4)

function GPU_VD()
    for i in 1:test_rounds
        sumdef = CUDA.zeros(Ny)
```

```
        @cuda threads=32 def_init(sumdef,tau,Y,alpha)

        temp = CUDA.zeros(Ny,Ny)

        @cuda threads=threadcount blocks=blockcount def_add(temp, P, beta, V0, Vd0, phi, Ny)

        temp = sum(temp,dims=2)
        sumdef = sumdef + temp
        for i in 1:length(Vd)
            Vd[i] = sumdef[i]
        end
    end
end

t_vd = @benchmark GPU_VD()
Times[1] = median(t_vd).time/1e9/test_rounds
println("VD Finished")


function GPU_Decide()
    for i in 1:test_rounds
        @cuda threads=threadcount blocks=blockcount Decide(Nb,Ny,Vd,Vr,V,decision,decision0,prob,P,Price,rstar)
    end
end

t_decide = @benchmark GPU_Decide()
Times[3] = median(t_decide).time/1e9/test_rounds
println("Decide Finished")

function GPU_Update()
    for i in 1:test_rounds
        err = maximum(abs.(V-V0)) #These are the main time consuming parts
        PriceErr = maximum(abs.(Price-Price0))
        VdErr = maximum(abs.(Vd-Vd0))

        @cuda threads=threadcount blocks=blockcount saxpy(Vd,Vd0,delta,1,Ny)
        @cuda threads=threadcount blocks=blockcount saxpy(Price,Price0,delta,Nb,Ny)
        @cuda threads=threadcount blocks=blockcount saxpy(V,V0,delta,Nb,Ny)
    end
```

```
end

t_update = @benchmark GPU_Update()
Times[4] = median(t_update).time/1e9/test_rounds
println("Update Half Finished")
println("Nb=",Nb,"Ny=",Ny)
println(Times)


function GPU_VR()
    for i in 1:test_rounds
        @cuda threads=threadcount blocks=blockcount vr(Nb,Ny,alpha,beta
    ,tau,Vr,V0,Y,B,Price0,P)
    end
end

t_vr = @benchmark GPU_VR() seconds = sec
Times[2] = median(t_vr).time/1e9/test_rounds

println("VR Finished")
print(dump(t_vr))
println("Update Fully Finished")
println("Nb=",Nb,"Ny=",Ny)
println(Times)
```

Code: Julia CUDA Benchmark Code

Benchmark performance

Julia vs Julia CUDA performance sheets are presented below:

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, namely by fitting `@timed` around each component in the implementation:

```
while (err > tol) & (iter < maxIter)
    Code for Initialization

    t = @timed begin
        Code for Value-of-Default
    end
    time_vd += t[2]
```

	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	0.000371494
	1000	/	10.3433655	0.012104929	0.000130831
	1250	/	/	0.014702145	0.000198855
	1500	/	/	0.018612573	0.000309957
	2500	/	/	0.030095512	0.000857451
	3750	/	/	0.04126148	0.000794312
	5000	/	/	0.049021429	0.000966725
	7500	/	/	0.076560898	0.001565692
	10000	/	/	0.100674108	0.002165904
	11250	/	/	0.10624682	0.000996565
	12500	/	/	0.12888245	0.00130683
	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 12: Julia vs Julia CUDA, Grid size 7*Nb

```

    t = @timed begin
```

	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 13: Julia vs Julia CUDA, Grid size Ny*Nb (Ny=Nb)

```
        Code for Value-of-Repayment
    end
    time_vr += t[2]

    t = @timed begin
        Code for Decision
    end
    time_decide += t[2]
```

```

    Code for Update
end

time_vd /= iter
time_vr /= iter
time_decide /= iter
```

Code: failed benchmarks

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

Next, we turn to measure components of the algorithm individually, we use timing Value-of-Repayment as an example.

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

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

For an individual component, directly benchmark the kernel:

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

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

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

Code: Julia CPU benchmarks which fail in Julia CUDA environment

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