# Parallel Computation of Sovereign Default Models[*]

A guide to parallel computation in Julia and CUDA

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

July 2, 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 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 tricks, and coding advice. The benchmark implementation in Julia with CUDA shows a substantial speed up over 1,000 times compared to standard Julia. 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 economic models. We choose the modern programming language Julia to construct 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

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 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 sovereign default model. The highly nonlinear Sovereign default model cannot be solved with fast methods like perturbation. A common approach is to use iterative 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, 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

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

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

[3] The latest stable version is 3.2.1.

language and the Julia-style design of CUDA library in Julia provide an efficient process 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 (2016) applied some of their insights to solve the canonical default model using C++ and Thrust.[5] More recently, Guerron et al. (2021) demonstrate that the C++/Thrust infrastructure can be used to speedup the estimation of nonlinear factor models using particle filtering. Khazanov (2021) uses C++/Thrust to solve a sovereign default model to study currency returns in emerging economies.

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. More recently, NVIDIA has 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 toolboxes needed to implement the sovereign default model in Julia CUDA. The code snippets will demonstrate how to use these tools. Since many online resources provide excellent CUDA tutorials[6], there is no need to reinvent the wheel. The paper does not intend to give a thorough tutorial into the world of CUDA or to emphasize low-level CUDA techniques for optimal performance. Instead, we address one central trade-off for economists: to compute highly complex economics programs unattainable by CPU with minimal coding effort. Under this guideline, the section is dedicated to introduce the selected basics of CUDA, with the end goal to help economists easily encapsulate complex economic problems into CUDA programming, and in particular, kernel programming.

### 2.1 NVIDIA CUDA in Julia

NVIDIA CUDA in Julia is part of the programming platform of JuliaGPU, which aims at unifying GPU programming in Julia. With high-level syntax and flexible compiler, Julia is well positioned to productively program hardware accelerators like GPUs without sacrificing performance. Indeed, perhaps the most persuasive argument to switch from the more mature C++ CUDA environment to Julia CUDA is the simplicity and flexibility offered by Julia NVIDIA CUDA.

NVIDIA CUDA is the best supported GPU platform among the JuliaGPU platforms (Innes (2020)). In NVIDIA CUDA, CUDA.jl package provides the programming support to use NVIDIA GPUs in Julia. It is built on the CUDA toolkit and aims to offer same level of performance as CUDA C. 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, providing a user-friendly yet powerful abstraction that does not require any GPU programming experience;

2. By writing CUDA kernels, with the same performance as kernels written in CUDA C;

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

[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

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

This paper focuses on the first two approaches, with extra emphasis on kernel programming, which provides an excellent combination of flexibility in programming economic models, a gentle learning curve, and ease in coding and testing the models.

## 2.2 Loop Fusion

Loop fusion provides access to the convenient Julia CUDA's linear algebra calculations, with speed on par to the well-known CUBLAS package for linear algebra operations in C++ CUDA. Instead of writing traditional vectorized loops, loop fusion provides a faster calculation without any overhead. The difference may be subtle in small scale calculations. However, when working with large arrays with big data, the overhead could be in the size of gigabytes, and execution will 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 the default model's utility calculation, loop fusion reduces execution time from hours by standard broadcast down to seconds.

A simple example of loop fusion with one matrix $X$ is shown below to evaluate $f(3*\sqrt{X}+4x^3)$

```
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 without requiring extra space 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 a small sets of operations to be fused, but Julia allow generic application even for user-defined array types and functions. This convenient feature, however, requires careful inspection, and will be discussed in the default model implementation section.

## 2.3 MapReduce

Mapreduce is a widely used method to process large data sets. Introduced by Google in "MapReduce: Simplified Data Processing on Large Clusters" (J Dean, 2008), it relieves the burden of programming details of parallelization and optimization by providing a simple interface. In Julia CUDA, `map` and `reduce` provide the high-order generic array operations. Highly extensible, Julia CUDA's map/reduce can be applied to all types of arrays, including the standard Julia CUDA array type, CuArray (Besard et al., 2018). Matrices stored in GPU are recommended to be handled by Mapreduce if applicable so that the operations can be parallelized for high performance.

Operations `sum` and `max` are common in sovereign default models, and their implementation plays an important role in optimizing the solution algorithm. For example, 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 get 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. Figure 1 illustrates the `reduce` operation on a matrix.

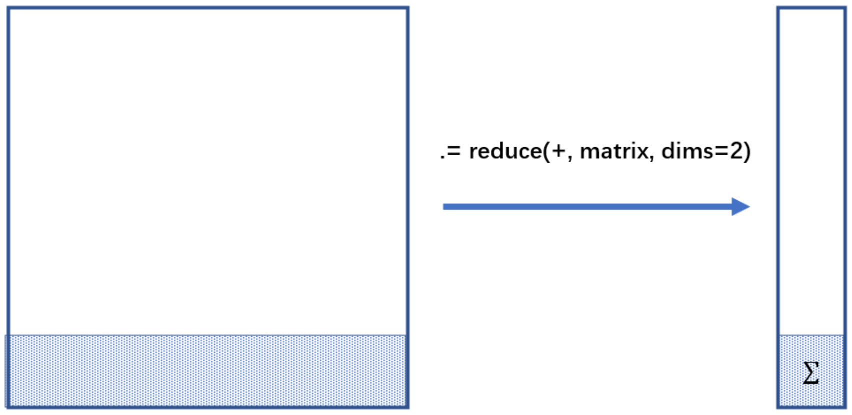


Figure 1: reducing matrix by row

In Julia, the implementation of `reduce` offers a fully generic interface that specializes on argument types (Besard et al., 2018), offering more types compared to only primitive types in C CUDA.

## 2.4 CuArray

The primary interface of data management in Julia CUDA is through `CuArray`. CuArray follows many characteristics of the standard Julia array. A brief introduction of the syntax is referenced below from official documentation of Julia CUDA (Besard (2016)). More details of data management will be discussed in Section 6.

```
# 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)
```

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. Compared to standard CPU functions which are executed on a few CPU threads, kernels are executed in parallel on thousands of GPU threads.

When implementing complex economics problems, designing good kernels offer the most direct and most significant improvement in performance. Thanks to the highly aligned syntax of standard Julia and Julia CUDA, economics problems running on standard Julia could often be easily transformed into a series of kernel functions executable on Julia CUDA. The speedup through kernels constitute the central performance improvement of our Sovereign Default Model implementation in Julia CUDA.

A central reason to program through kernels in Julia is the exceptional speed performance. Julia on CPU is known for computational speed comparable to C, and the same holds for Julia on GPU with kernels written using CUDA.jl. The Julia CUDA's official benchmark (figure 2) shows how specifically designed Julia kernels approach and exceed performance of CUDA C kernels Besard et al. (2018).

In this subsection, we first give a brief overview of the GPU architecture on a hardware level, then proceed to show basic kernel programming in Julia CUDA, and finally we construct a simple kernel function that increments all elements of a matrix by one.

We begin with a very short introduction of GPU hardware design to provide helpful intuition for the art of designing kernels. The basic unit for computation in a GPU is a processor core. Each processor core is a stream processor capable of running an instruction for one thread at a time. In a GPU, processor cores are organized into stream

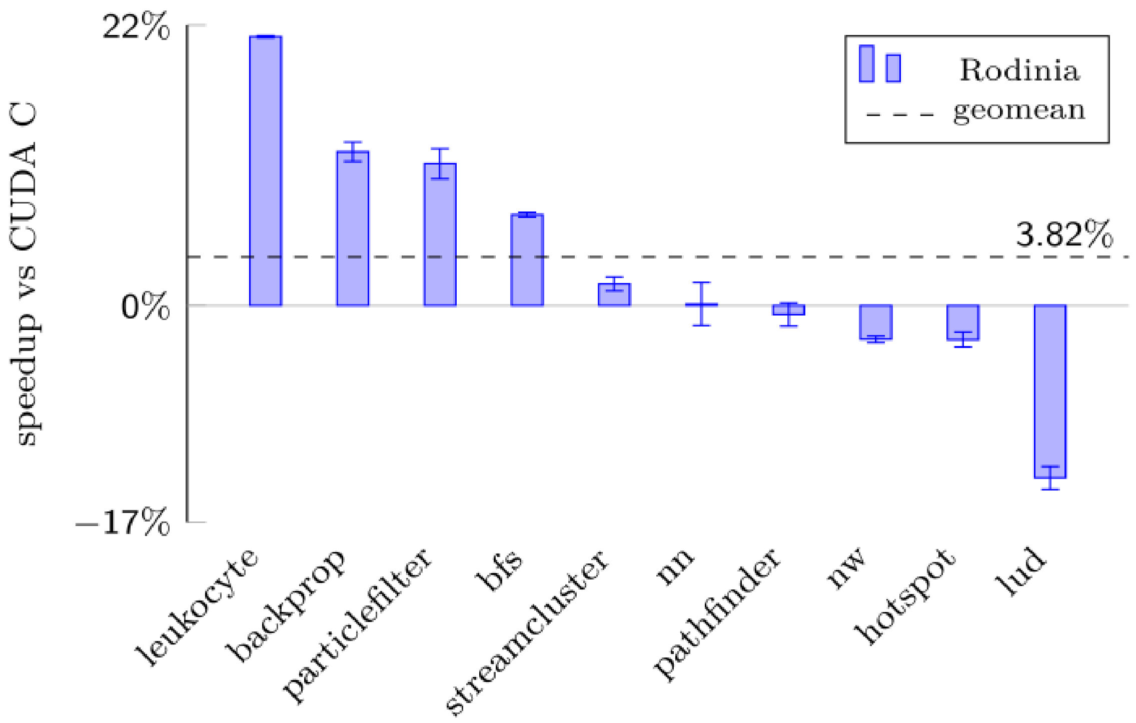
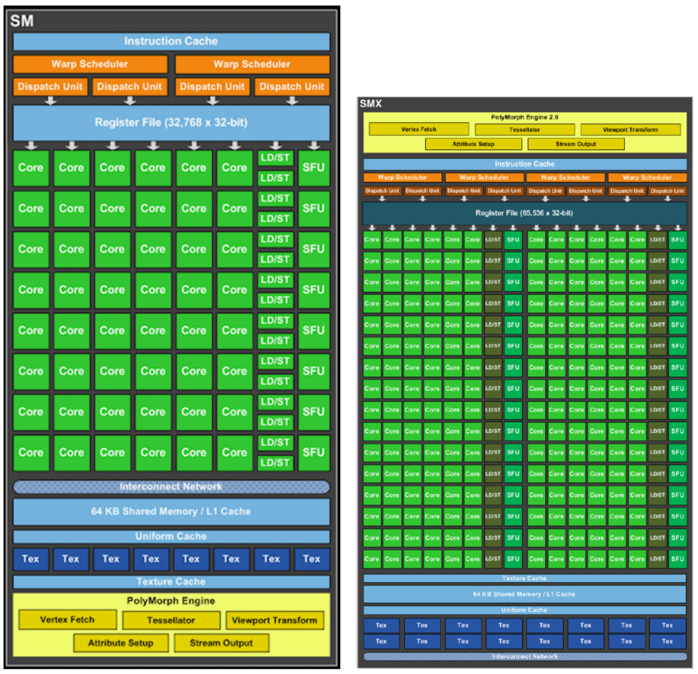


Figure 2: Relative performance of Rodinia benchmarks implemented in Julia with CUDA.jl

multiprocessors. Figure 3 shows the compositions of stream multiprocessors (SM) in GPU of the old Fermi architecture and new Kepler architecture. (Letendre (2013)).



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

Figure 3: Comparison of Fermi and Kepler stream multiprocessors

Theoretically, CUDA identifies the processor cores through an index system of threads, blocks and 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 is 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 4 (Innes, 2017).

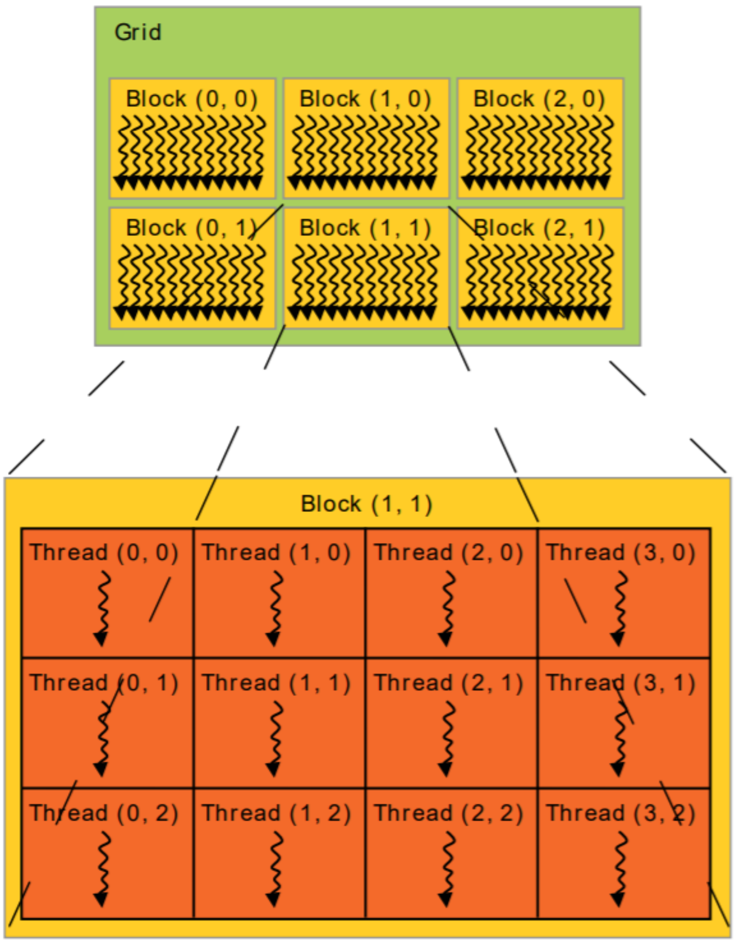


Figure 4: Grid of thread blocks

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
```

Back to 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 take hardware limit into consideration when assigning the size of a thread block. We recommend using a $32 \times 32$ thread block.

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 matrix, with each thread uniquely identified to a particular state of the double for-loop and computing the output at that state.

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 is each decremented by 1, since Julia uses 1-indexing instead of C's 0-indexing.

To loop through each variable within the range desired, in this case $[1, Nx]$ and $[1, Ny]$, we only execute the thread when its index passes the following conditional statement:

```
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
```

It is worth noting the ease to modify the original double-loop in standard Julia to a kernel function in Julia CUDA. The sole effort lies in designing the dimensions of threads to fit the nature of the problem. Behind the apparent simplicity lies the limits of kernel programming. We will address that in Section 6.

After finishing the kernel design, we proceed to executing the kernel. First we assign the sizes of the dimensions of a thread block and the grid. When every loop is large, assigning size of $(32, 32)$ to a thread block is a convenient practice. The following snippet shows assigning thread blocks (threadcount) of size $(32, 32)$ and the corresponding size of grid (blockcount). The third line demonstrates how to execute the kernel by providing the threatcount 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)
```

In the appendix, we included the entire set of code for our model in CUDA.

## 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.
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 $\tau$ is the fraction of the endowment lost because of default and $\phi$ 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 (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 (Guerron, 2016):

**Algorithm 1:** Psudocode Defualt Model in Standard Julia

```
Initialization;
Define maxind = $N_y \times N_b$;
while $error > tol$ and $iter < maxiter$ do
    $\hat{V} = V$;
    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:
        $V_d(i_y) = U((1-\tau)y(i_y) + \beta E[\phi V(y', 0) + (1-\phi)V_d(y') \mid y(i_y)]$;
        $V_r(y(i_y), b(i_b)) = \max_{b'} U(c) + \beta E[V(y', b') \mid y(i_y), b(i_b)]$ subject to
          constraint $c \geq 0$;
        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: $q(y(i_y), b(i_b)) = \frac{1-prob(default)}{1+r^*}$;
    end
    $error = \max((\hat{V}, V))$;
    update Value matrix: $V = \delta V + (1-\delta)\hat{V}$;
    iter++;
end
```

## 4 Sovereign in Julia

This section shows the key points in the computational implementation of the sovereign default model. The implementation is divided into three parts: Value-of-Default, Value-of-Repayment, and Decision. To illustrate the practical implementation of the key Julia-style improvements discussed in Section 2, we will compare the standard Julia and Julia CUDA side by side. A noteworthy point to highlight is how subtle the differences are between the standard codes and the CUDA counterparts. A central reason for choosing Julia CUDA for large scale economics modeling is this smooth "upgrade" of code from CPU infrastructure to GPU infrastructure in the Julia environment. This means that a prototype solution can be quickly implemented and tested in standard Julia, and later be tweaked to large-scale GPU computation efficiently. Our design builds on Guerron's implementation with Thrust in C CUDA, but we will show how the Julia environment benefits economists to code efficiently and concisely through the techniques discussed in Section 2.

### 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 implementation is to run through a double for-loop:

```
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: instead of running the double loop directly, we can choose to store $f(y|iy)$ for each pair of possible values for $(y, iy)$ in a temporarily initialized matrix and reduce along the columns. The benefits and costs of using temporary matrix to facilitate calculation will be discussed in the code design section.

```
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)
```

Loop fusion: Given the value function

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

, an alternative method is through loop fusion. Recall that loop fusion provides a significant speed up just by "adding a few dots." In this specific implementation, 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
```

## 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 with computation complexity $O(n^4)$.

In our CPU design, Julia-style linear algebra and loop fusion operations again replace the standardized for-loops. In the CUDA implementation, we use division of kernels to provide simple and efficient GPU computation by reducing synchronizing cost and temporary matrix allocation, two banes common to large-scale economic model simulations.

```
for b in 1:Nb
    c = exp(Y[iy]) + B[ib] - Price0[iy,b]*B[b]
    if c > 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
```

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

A simple and efficient fix is to divide the value-of-repayment calculation into components. Each component is computed in a compact kernel, minimizing synchronization cost per kernel. The code for the three kernels are attached in the appendix and GitHub.

Since the `'` operator is not currently supported by Julia CUDA in kernel computations, the calculation of the sum-of-return requires manually implementing one additional for-loop, making a total of four for-loops. Two for-loops are reduced by the two-dimensional thread assignment, and two loops are contained in kernel calculations. This reduces the complexity from $O(n^4)$ to roughly $O(n^2)$, a big speed up to about twenty-five hundred times faster even on a rough $50 \times 50$ grid (the maximum speed up is bounded by the GPU design). In the bench-marking section, the two-for-loop design will illustrate its satisfying result to a speed-up over ten thousand times.

## 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.[7]

### 5.1 Julia CUDA

The benchmark analysis is performed on the three main components of the model's computation: Value-of-Default, Value-of-Repayment, and Decision. Figure 9 plots the

[7]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.

results from Julia CUDA benchmark. The X-axis shows the size of grid points for the Endowment×Bond Matrix of size $Ny \times Nb$, and the Y-axis shows the median running time in microseconds for one evaluation of each component of the model. One purpose of this benchmark parametrization is to maximize the difference in time complexity 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 ($n = Ny = Nb$). Therefore, the Value-of-Repayment component will dominate runtime, and that is where Julia CUDA provides the greatest speedup. Later in the section, we will also provide a more realistic parametrization of the sovereign default model. The time complexity of each component is summarized as below:

| 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

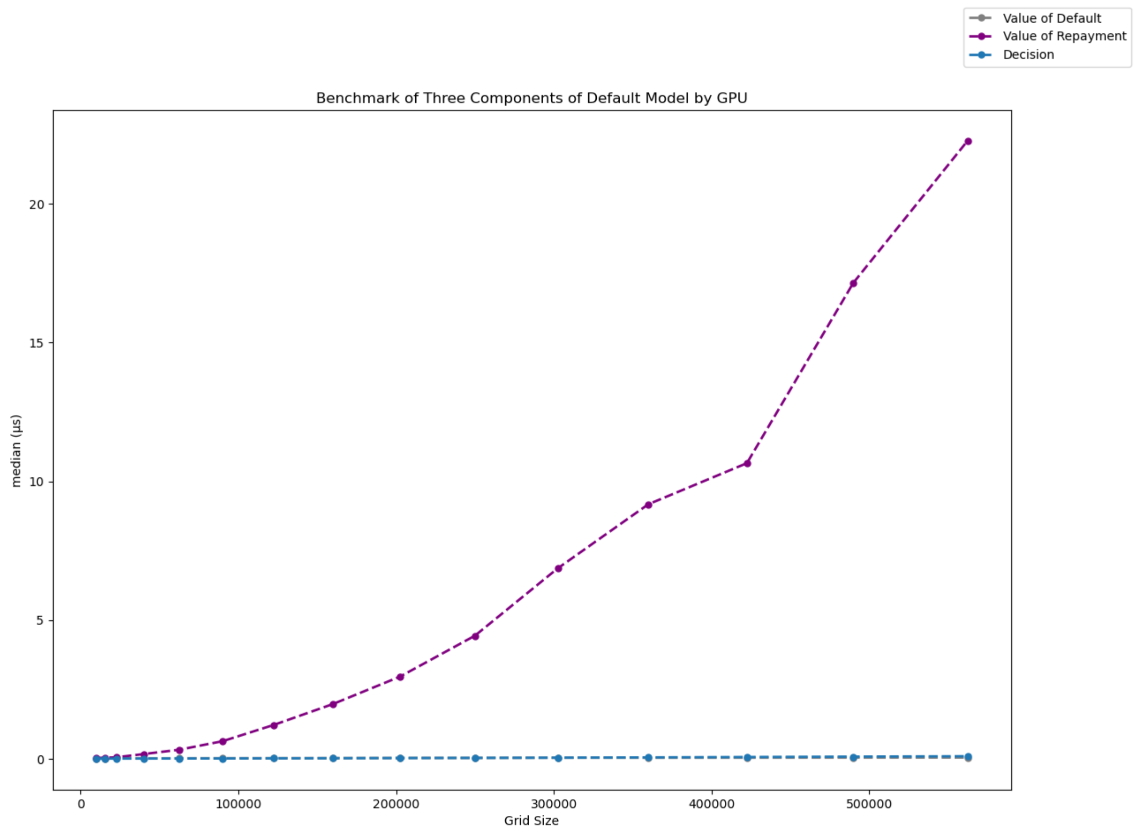


Figure 5: GPU Benchmark of sovereign default model

The figure showcases the performance under grid granulation from $100 \times 100$ up to $500 \times 500$ grid points for the Endowment $\times$ Bond matrix. The increase in running time shows a weak polynomial factor but follows more a linear trend. This indicates the GPU processor has not reached its physical limit even with a large grid. The computation is extremely fast even for the grid size of $640,000$ points. The major bound to our benchmark comes from the memory usage. For example, to store a single Cost Matrix with $Ny = Nb = 500$ requires $500^3$ of Float32 memory, 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. [8] Overall, the first benchmark demonstrates that Julia CUDA model can run extremely efficiently under reasonable and desirable granulation of the state space. We proceed to compare Julia CUDA's performance to standard Julia on CPU, and Stndard Parallel Library with C++.

## 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. The ease to transition between CPU design to GPU design in Julia is one of Julia's top advantages. Due to the close resemblance of the two models, the benchmark comparison offers a clear demonstration of the speedup achieved by GPU acceleration.

In this section, we provide two benchmark tests. The first benchmark follows a standard parametrization of 7 endowment points with varying numbers 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 runtime is dominated by the Value-of-Repayment component, which has complexity $O(n^2)$. We ignore the Value-of-Default part given its constant complexity.

The second benchmark is performed on a endowment$\times$debt grid of size size $Ny \times Nb$ $, Ny = Nb$, and the time complexity is 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 3: Time Complexity of the Second Benchmark

The following graph demonstrates the dominating performance of GPU given much greater challenge in complexity:

[8]A quick fix of the memory issue is to remove the high-dimensional matrix that had speed up the calculation. In the trade-off of memory for speed, the standard vectorized loops replaces 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.

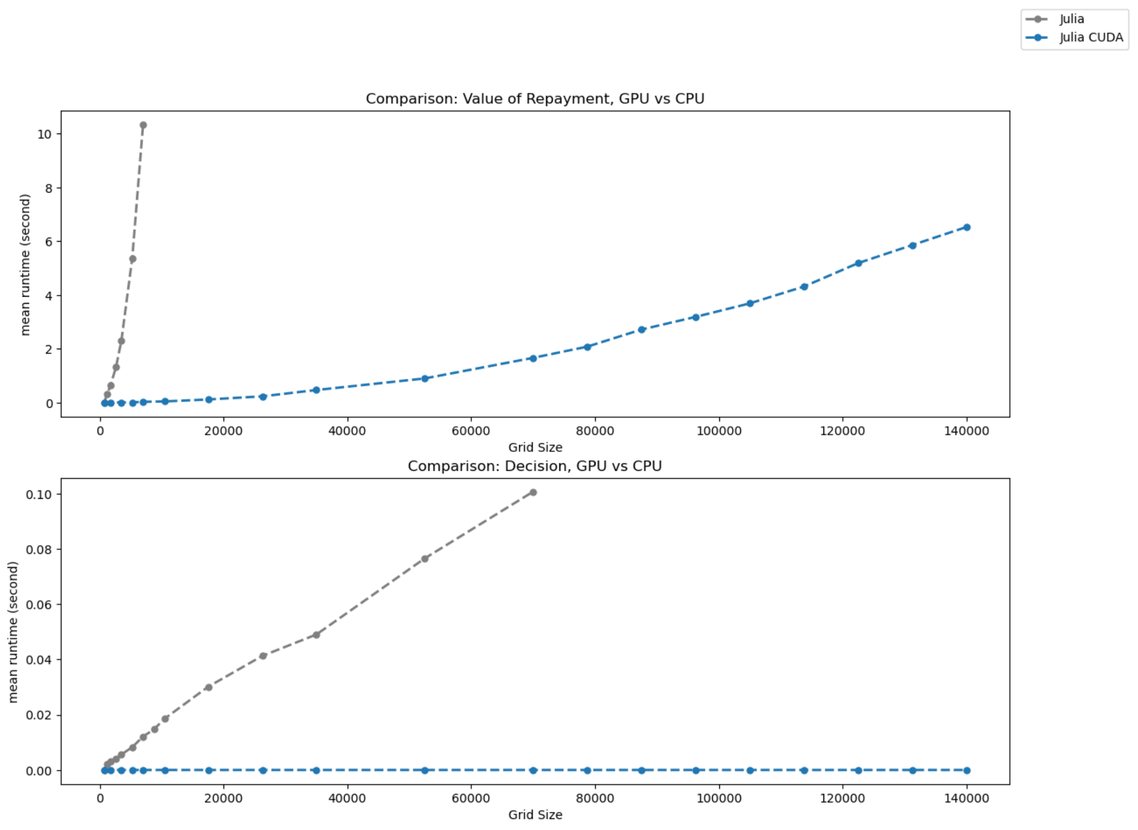


Figure 6: First Benchmark of grid size $7 \times Nb$. Comparison of the three components in the model

The program's runtime performance is dominated by the most time-consuming process, Value-of-Repayment calculation. The Value-of-Repayment calculation has complexity of $O(Ny^2 Nb^2) = O(n^4)$ for Endowment $\times$ Bond matrix of size $n \times n$. For CPU computation, runtime of value-of-repayment calculation grows in polynomial scale to over 10 seconds on a $100 \times 100$ 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 $0.0467s * 500 = 0.39$ minute.[9]

## 5.3 Julia CUDA vs C++ stdpar CUDA

In this subsection we compare the computation speed of Julia CUDA versus that of the C++ CUDA standard parallel library (Stdpar) implementation. One noteworthy fact is that Stdpar's Value-of-Default component performs better than the counterpart in Julia CUDA.

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

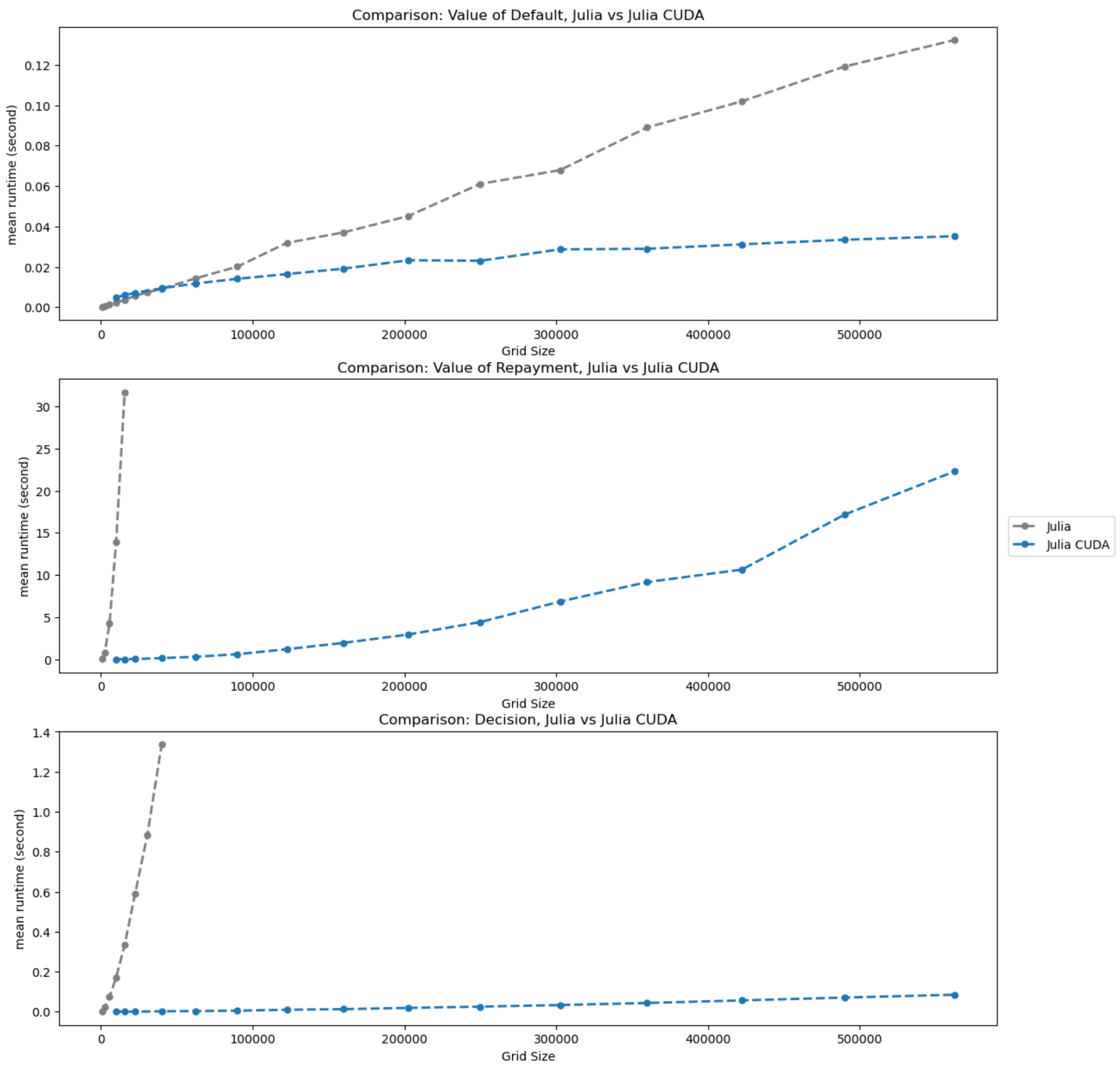


Figure 7: Second Benchmark of grid size $Ny \times Nb, Ny = Nb$. Comparison of the three components in the model

An important point to address is that our Julia CUDA and Stdpar C++ implementations are different in some of the code designs. Our model implementation with C++ Stdpar 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. Deficiencies also arises. Inefficiency on large data grid is unavoidable since an oversized class instance stores the entire model with the data and functions. Even with these limits, the Stdpar C++ design is fast, though exceeded by the Julia CUDA implementation.

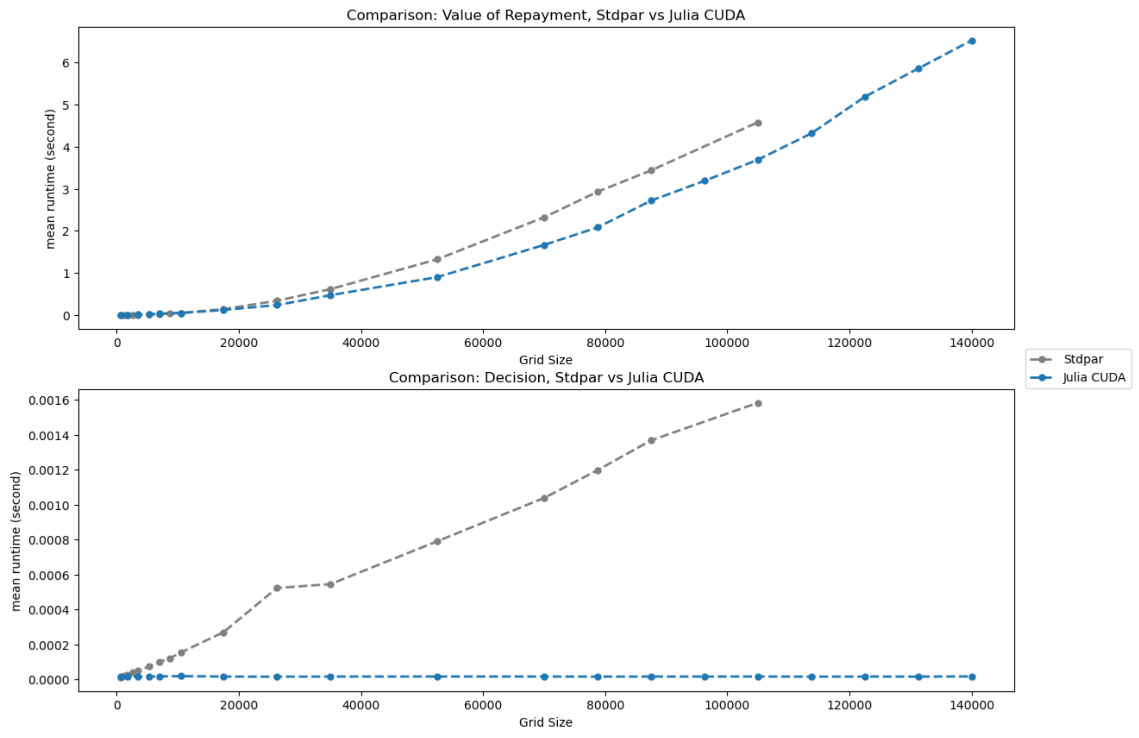


Figure 8: First Benchmark of grid size $7 \times Nb$. Comparison of the three components in the model.

## 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 proceed to investigate the potential fluctuations of the kernels' performances in real-time. We do so by sampling the distributions of run-time for the three major kernels that compute Value-of-Default, Value-of-Repayment and the Decisions. Any outlier could lead to a significant slow down in performance if the same delay occurs in multiple iterations in execution. That is when the model is solved via value function iteration.

We present boxplots for two sets of parameters of $(Ny = 100, Nb = 100)$ and $(Ny = 7, Nb = 1000)$. 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 Julia of which the kernel takes extra time for pre-compilation in the first round. Performance of the kernels after the first iteration are shorter in run time and display 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, the performance of Julia CUDA in running the sovereign default model will be relatively stable in real-time.

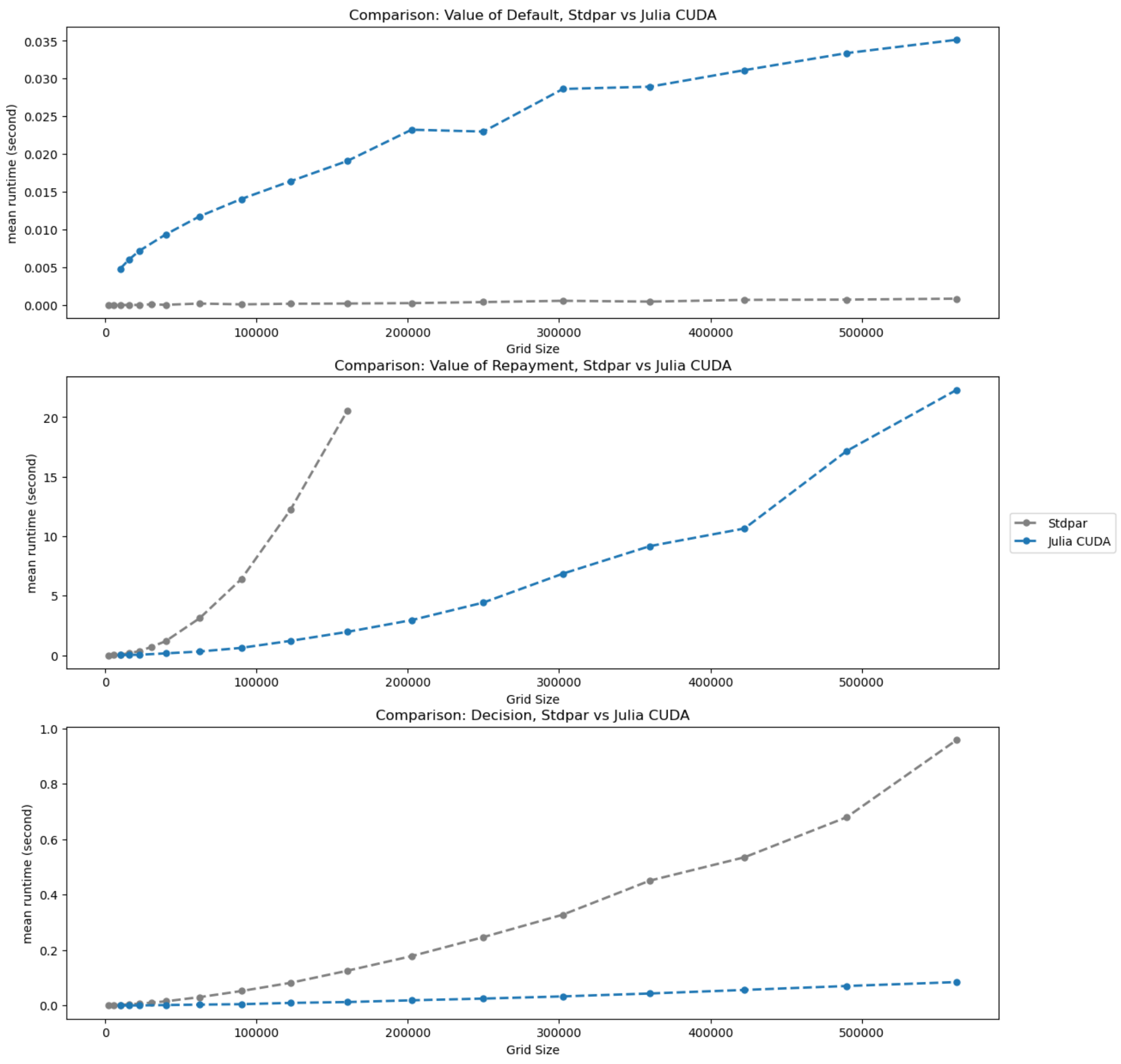


Figure 9: Second Benchmark of grid size $Ny \times Nb, Ny = Nb$. Comparison of the three components in the model

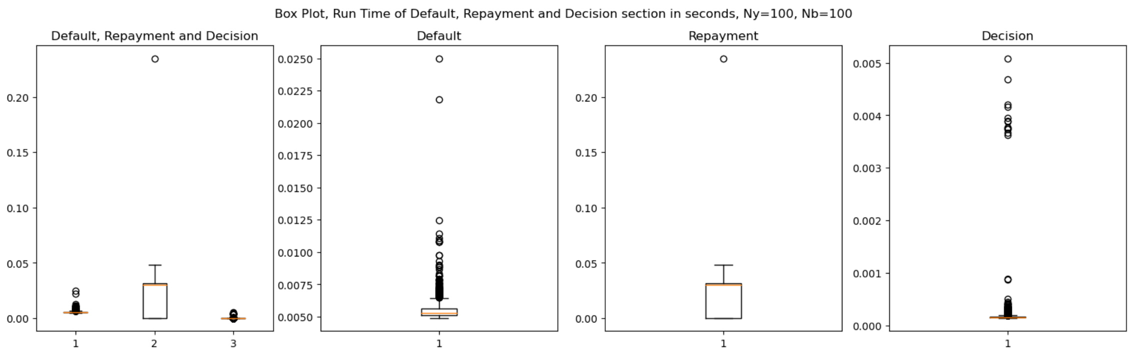


Figure 10: Ny=100, Nb=100

## 5.5 Actual Run-time

This section compares the actual execution time for the solution of the sovereign default model using algorithm 1 and the programming languages: Julia, C++, Julia CUDA and

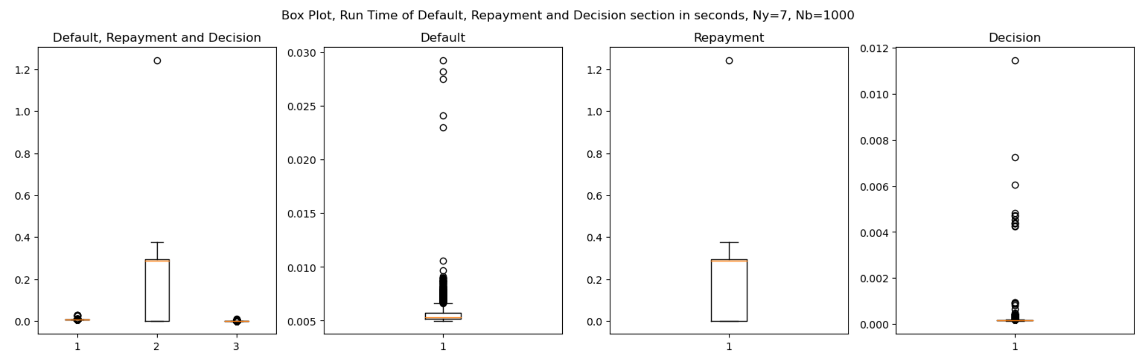


Figure 11: Ny=7, Nb=1000

C++ Standard Parallel Library. 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 see that Julia is almost as fast as C++. 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.

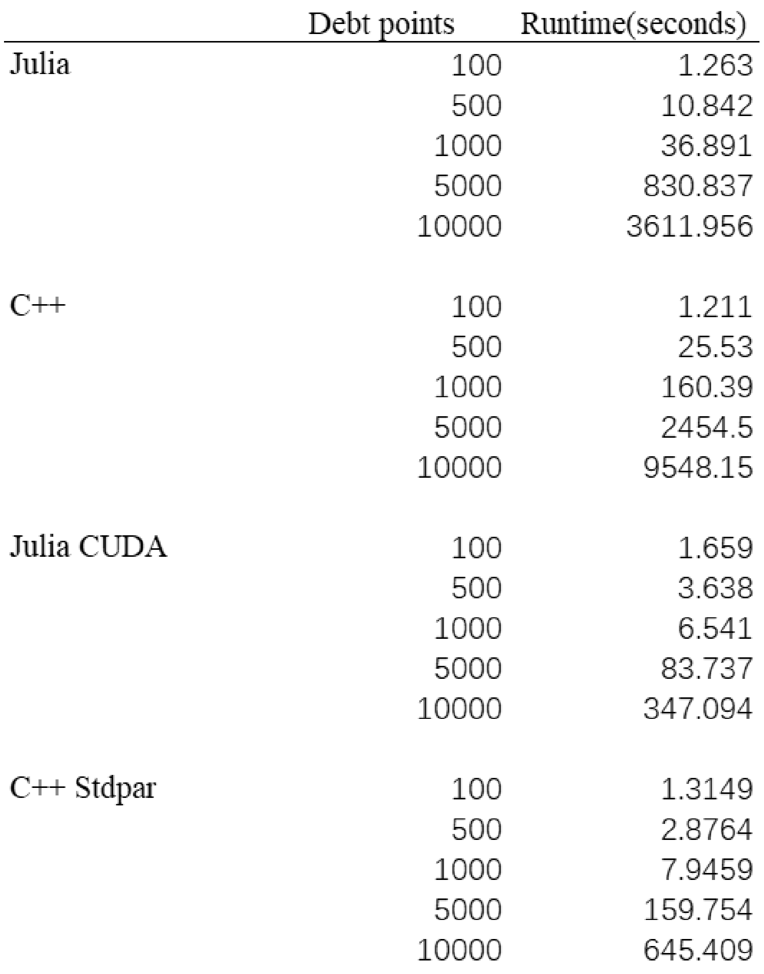
Julia CUDA dominates the benchmarked implementations. At $7 \times 10,000$ points, Julia CUDA executes the program 10 times the speed of standard Julia. A huge cost comes from overhead and memory allocations, as the computation speedup of Julia CUDA compared to Julia in the main model components exceeds 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`. In addition, we point out our many failed attempts of benchmark in the appendix.

We present our final benchmark method. We use 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 for a number of times. Then we test `@benchmark` on the new function. 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)
```



| | 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 12: Speed Comparison

```
    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
```

In the appendix, we present a series of benchmarks that would fail to capture the correct time in Julia CUDA. A simple sanity check is by comparing the performance of Value-of-Repayment to that of other kernels. If Value-of-Repayment gives unbelievably quicker results than other components, there is often an error with the benchmark.

## 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 model's implementation. Some advanced features in C++ CUDA are suggested for future developments of Julia CUDA. We want to point out how the advice differs from other languages like standard CUDA with C++ or standard Julia.

### 6.1 Synchronization

In this section we give a detailed explanation for dissecting the sovereign default model algorithm into multiple kernel functions. We recommend such modular design for coding CUDA in Julia both for coding efficiency and for future transfer of coding design across programming platforms.

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, that is to say 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 require a more complex control flow that requires communication across threads.

The control flow of economic models often require a thorough update of values before proceeding to the next phase of calculation. 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 all 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 time spent waiting for slower threads to finish and then to synchronize the updated information is called the synchronization cost. Synchronization cost can be divided into two components: the maximum synchronization cost from the wait for data access, and the minimum synchronization cost from waiting for the slowest warp to finish. For our purpose to introduce efficient CUDA implementation, the simplest choice is to manually divide the algorithm into multiple kernels.[10] Such division has multiple benefits:

1. Compared to a single kernel function, run-time variance among kernels is drastically reduced, as demonstrated in the run-time distribution.

2. Data transfer for computation is reduced in each step, thus easing storage limits of GPU. For large scale economic computation with heavy data, such improvement is crucial.

3. Debugging and feature-adding are much more efficient in a modular design.

## 6.2 Memory Management through Garbage Collection

With the objective to improve performance of data-heavy economic models running in Julia CUDA, this subsection extends the discussion of CuArray to introduce details about garbage collection according to Julia CUDA's official manual.

Juia 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 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.

Avoiding garbage collection could significantly improve performance speed, as GPU has smaller memory and frequently run out of it. Calling `unsafe_free!` method allows manual memory free up without depending on the Julia garbage collector.

Batching iterator provides an interface for economic models with real heavy data input beyond the capacity of GPU. An official example of using `CuIterator` to batch operations is provided below (Besard (2016)):

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

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

```

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

```
Batch 1: [3]
Batch 2: [7]
```

Optionally, to improve speed, use `unsafe_free!` to manually free up batched memory after usage before waiting for the garbage collector.

## 6.3 Temporary Matrix

Allocating temporary matrix during initialization is a double-edge sword that requires close examination. Temporary matrices provide an optional trade-off to increase speed at the cost of extra memory allocation on GPU. The primary reason for assigning temporary matrices on GPU is to avoid memory access cost.

For instance, in the appendix, we included calculation of Value-of-Repayment through temporary matrices. Several temporary matrices were created to facilitate the calculation. For example, Cost matrix $C[iy, ib, b]$ 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 three-folds. Indeed, our standard implementation for Julia CUDA does not utilize temporary matrices. Firstly, temporary matrices could be very large. Take the cost matrix $C[iy, ib, b]$ for example. Suppose there are 7 endowment levels with 10000 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 important 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 host CPU, but run kernels on GPU with minimal data transfer to host CPU. But then the first disadvantage could not be avoided if the memory is stored on GPU.

Continue with the example of 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, which is paired with 6GB GDDR6 memory. A single 5.215 GiB temporary matrix would be dangerously close to a memory overflow. If the matrix is instead stored on host CPU, for each of the three hundred rounds of value iteration, the 5.215 GiB matrix must be transferred from host CPU to device GPU and back again. If theoretical transfer speed is 16GiB/s, the transfer cost for a single temporary matrix would already take more than $5.215/16 \times 2 \times 300 = 195$ seconds. With more temporary matrices, the data transfer cost becomes highly undesirable. [11]

## 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 avoiding allocating and de-allocating temporary arrays should each broadcast be executed sequentially. However, automatically fusing multiple broadcast operations does not show the level of improvement as expected. The speed up of loop fusion diminishes when too many loops are fused together.

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.[12]

[11] 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

[12] 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,

The complexity of control flows in a model could escalate the challenge of both kernel design and run-time performance. GPU is most powerful for running the SIMD (Single Instruction Multiple Data) operations. However, CUDA performs much worse if 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 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.[13]

The limit of memory transfer bandwidth, as pointed out in the temporary matrix subsection, could lead to a slowdown in runtime performance. If we could not hold all data in GPU memory, we are required to copy the memory from CPU to GPU, and feed the updates back from GPU to CPU. The runtime transfer cost could dominate the actual computation cost in GPU. In the memory management subsection, we introduced one method in Julia CUDA 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 power of Julia CUDA is not so straightforward and intuitive. Complexities in control flow and memory transfer require scrutiny and coordination in code designs. However, performance-wise, these challenges are not limiting problems compared to the greater benefits of Julia CUDA programming.

## 7 Concluding Remarks

Motivated by ever more complex and improved algorithms in nonlinear quantitative models, this paper provides a framework to parallel computation in Julia. The choice of language emphasize our belief in Julia as a developing numerical language. This modern language provides a superior balance between execution speed and coding convenience. Compared to Python/C++, Julia's syntax follows many mathematical traditions, such as 1-indexing instead of 0-indexing, providing an exceptional experience in coding economic models.

we do not go into details about these interesting fields.

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

This paper focuses on computational details and coding standards for parallel computation in Julia. Using sovereign model as an example, we illustrate the strength of parallel computation in iterative models with knife-edge decision rules. We demonstrate Julia's capacity to write simple parallelization codes to cut down execution time in realistic models like sovereign default models. The performance of computation is critical particularly for large-scale economic models, and a combination with cloud computing and functional programming will provide a more efficient environment to write and test large-scale economic models.

## 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.*

Guerron, P. (2016): "Sovereign Default on GPUs," *Working paper Boston College.*

Guerron, 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

### 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
```

2. Value-of-Repayment

```
#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), then add   *sumret
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
```

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
```

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)
```

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
```

## 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

    #@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
```

```julia

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)
```

## Benchmark performance

Julia vs Julia CUDA performance sheets are presented below:

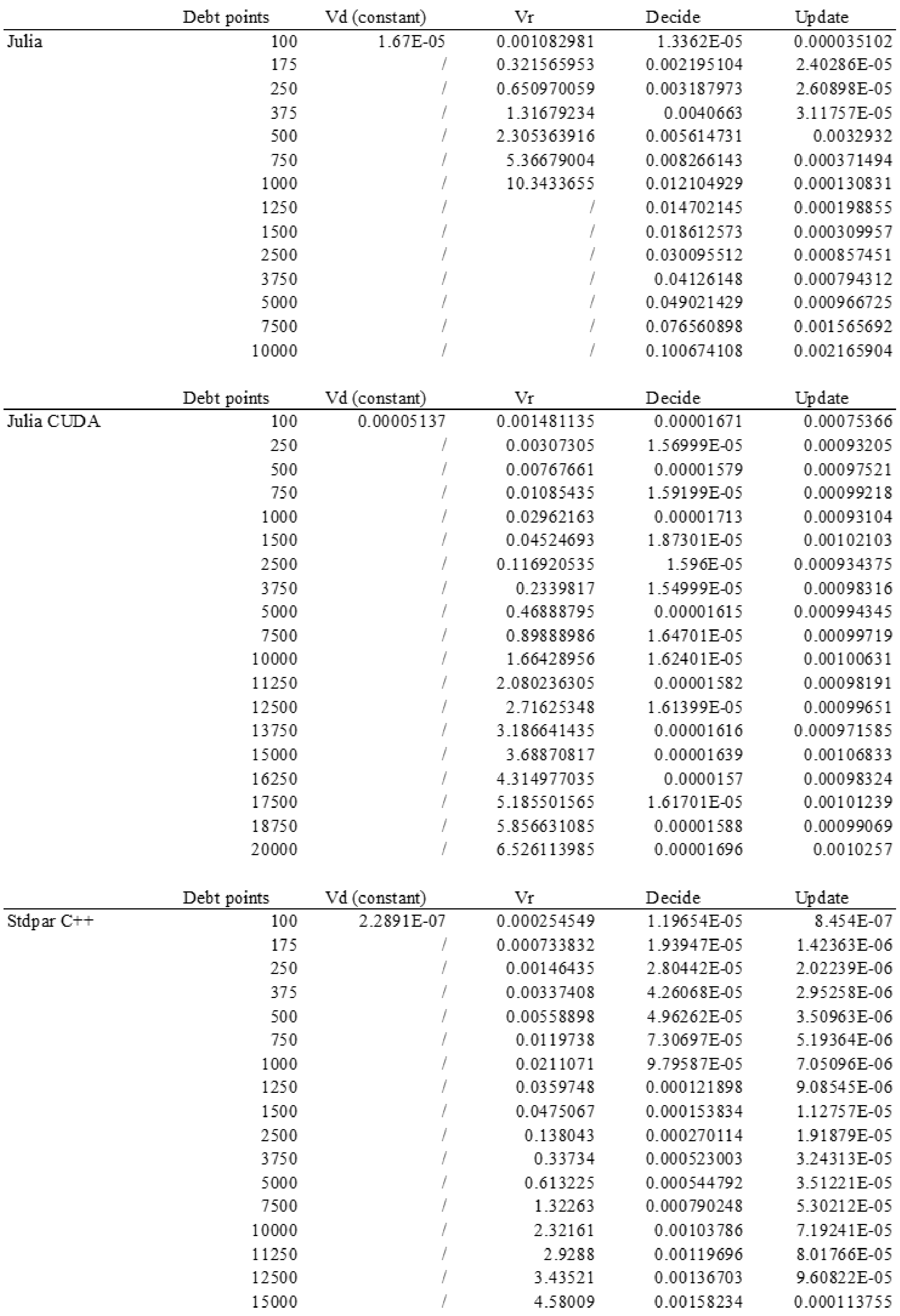
## Failed Benchmarks

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]

    t = @timed begin
        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
```

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.

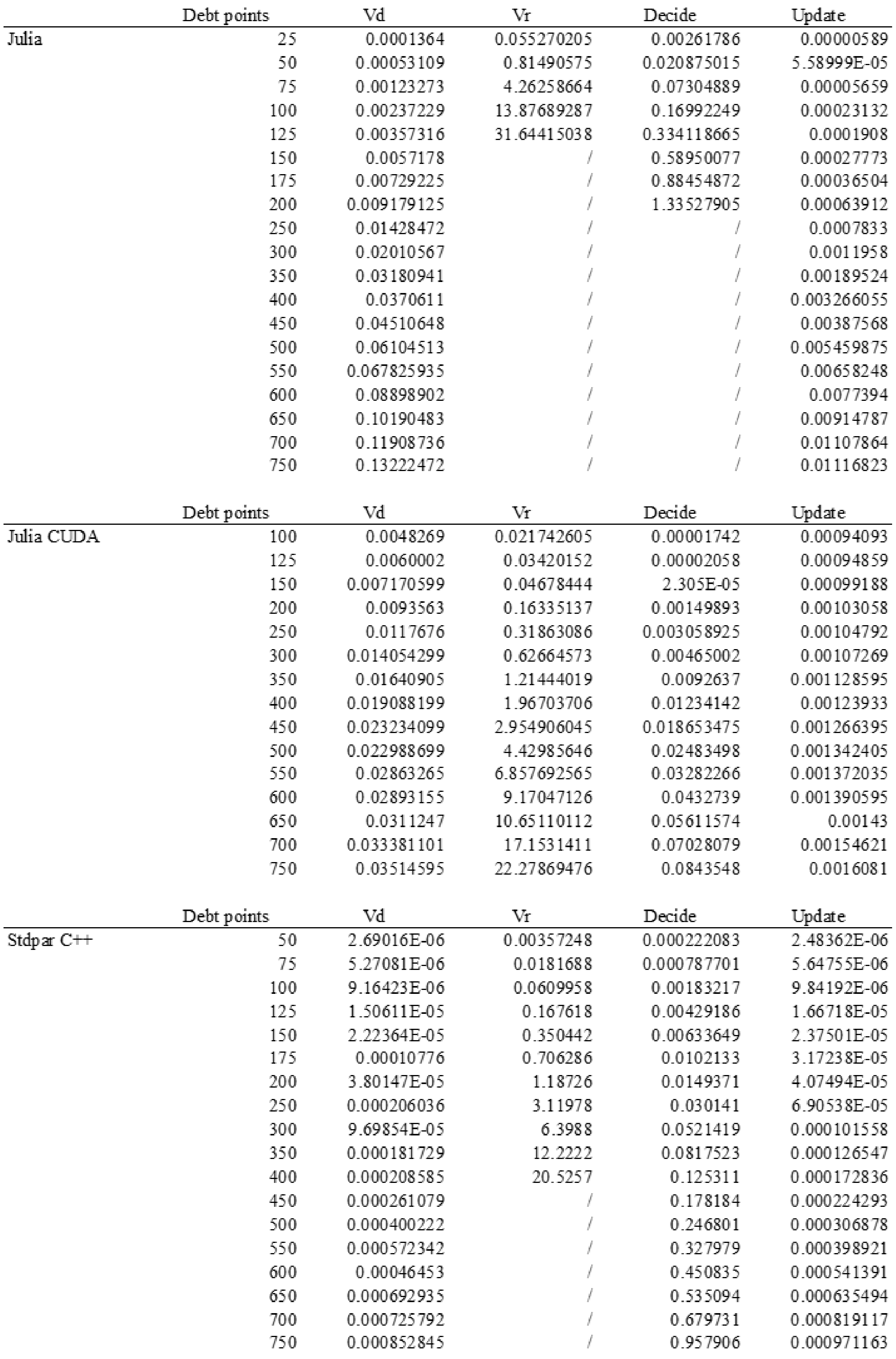


| | Debt points | Vd (constant) | Vr | Decide | Update |
|---|---|---|---|---|---|
| Julia | 100 | 1.67E-05 | 0.001082981 | 1.3362E-05 | 0.000035102 |
| | 175 | / | 0.321565953 | 0.002195104 | 2.40286E-05 |
| | 250 | / | 0.650970059 | 0.003187973 | 2.60898E-05 |
| | 375 | / | 1.31679234 | 0.0040663 | 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 |

| | 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 |

Figure 13: Julia vs Julia CUDA, Grid size 7*Nb



| | 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 14: Julia vs Julia CUDA, Grid size Ny*Nb (Ny=Nb)

Next we turn to measure components of 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
```

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