Sovereign Default on GPUs*

Pablo A. Guerron-Quintana†

April 21, 2016

Abstract

Preliminary

Solving default models are computationally expensive. This note shows how to solve the standard sovereign default model with the help of Graphics Processing Units (GPUs). The solution is at least 4 times faster than the version coded in C++. The improvement increases substantially with the number of grid points in debt.

Keywords: Sovereign Default, fast computation, parallelization, C/C++/CUDA

JEL classification numbers: F34, F41, F44

1 Introduction

Sovereign default models are highly non-linear models due to its knife-edge decision rules. As a consequence, fast solution methods such as perturbation are ruled out. The alternative, value function iteration methods, turns out to be slow and suffers from the curse of dimensionality.

This note shows that solving the sovereign default model using GPUs and CUDA can speed up significantly the solution. Moreover, it allows the researcher to use fairly populated grids that would be prohibitively expensive in languages such as Fortran or C++. I proceed in two steps. First, I discuss how to solve the model using C++ and then CUDA. The first step is necessarily because an important fraction of the C++ language is embedded into CUDA. Moreover, the absence of a native matrix object in CUDA (and C++) implies heavy use of 0-indexing vectorization, which is easier to implement in C++. In the following sections, it is assumed that the reader is familiar with sovereign default models (Eaton and Gersovitz

*I thank Juan Rubio-Ramirez for introducing me to parallel computing and guidance and Grey Gordon for useful comments. The code is available at my website All errors are mine.

†Federal Reserve Bank of Philadelphia, pguerron@gmail.com.

(1981)), their solution methods (Hatchondo et al. (2010)), and parallel programming (Kirk and Hwu (2013)). This note complements the recent review of computing languages to solve the neoclassical growth model in Aruoba and Fernandez-Villaverde (2015).

2 Model

Let's consider the standard sovereign default model (Arellano (2008)).[1] There is a central planner of a small open economy. Every period the planner decides whether to repaid its obligations (with value V_r) or default (with value V_d) to maximize welfare. That is, the planner makes her default decision based on $V(y,b) = max_{d\in\{0,1\}}\ (d\ V_d(y) + (1-d)\ V_r(y,b))$. If the sovereign is in good standing (d = 0), then her problem is

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

subject to the budget constraint

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

Here, $q(y,b')$ is the price of debt issued today and y is an exogenous endowment. In contrast, if the planner decides to default (or the country is excluded from financial markets), the country's value function is

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

Here, τ is a default cost (a fraction τ of output is lost) and ϕ is the exogenous probability of being readmitted to financial markets next period. For the numerical implementation, we use the following functional forms and parametrization: $U(c) = \frac{c^{1-\sigma}}{1-\sigma}$, $y = \exp(\tilde{y})$, where $\tilde{y} = \rho\tilde{y}_- + \sigma_y$, $\beta = 0.953$, $\sigma = 2$, $\rho = 0.9$, $\sigma_y = 0.025$, $\tau = 0.15$, and $\phi = 0.28$. The stochastic endowment process is discretized using Tauchen's method. Note that this parametrization is intended for illustration purposes. The high value of the discount factor implies counterfactually low default rates.

[1] In Gordon and Guerron-Quintana (2016a), we incorporate capital and labor choices to the standard default model and show that it can account for the business cycles properties of several emerging economies.

3 A Detour

A major difference between C/C++/CUDA and other scientific languages (such as Fortran and Matlab) is that matrices are not native objects in C. This complication makes trickier the transition to C-type coding. One alternative is to use third party libraries (such as Ron Gallant's SCL or Armadillo) but this comes at the expense of diminishing portability from C to CUDA. The second alternative is to input the model in vector format. This option is more time consuming in the short run but will pay off overtime.

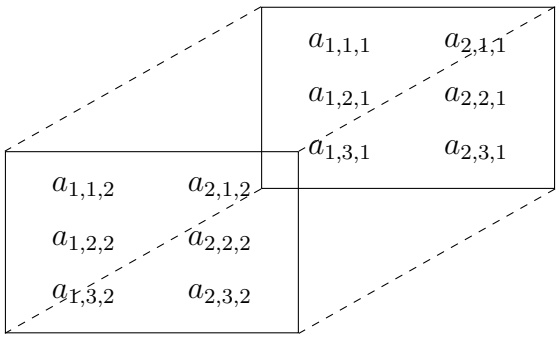
Writing mutidimensional arrays in vector format in C requires the use of row-major ordering, which consists on putting consecutive elements of the rows of an array in a continuous allocation of memory.[2] Let's consider a conventional 2×3 matrix

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{pmatrix}.$$

In row-major indexing, this matrix is transformed into the vector

$$Av = \begin{bmatrix} a_{1,1} \\ a_{2,1} \\ a_{3,1} \\ a_{1,2} \\ a_{2,2} \\ a_{3,2} \end{bmatrix} \begin{matrix} (0) \\ (1) \\ (2) \\ (3) \\ (4) \\ (5) \end{matrix}.$$

Next to the vector is the (*0-based*) indexes used to reference each of the elements in the matrix. Clearly, there is a unique mapping between the index and the matrix objects. Let's consider a bit more complicated example, i.e. a 3-D matrix:



Its row-major ordering version is

[2] Fortran, Matlab, and Julia use colum-major ordering

$$Av = \begin{bmatrix} a_{1,1,1} \\ a_{2,1,1} \\ a_{3,1,1} \\ a_{1,2,1} \\ a_{2,2,1} \\ a_{3,2,1} \\ a_{1,1,2} \\ a_{2,1,2} \\ a_{3,1,2} \\ a_{1,2,2} \\ a_{2,2,2} \\ a_{3,2,2} \end{bmatrix} \begin{matrix} (0) \\ (1) \\ (2) \\ (3) \\ (4) \\ (5) \\ (6) \\ (7) \\ (8) \\ (9) \\ (10) \\ (11) \end{matrix}$$

Once again, there is a unique link between each cell in the matrix and its vector equivalent. In general, let's consider a matrix of T dimensions with each of its elements represented by the *t-tuple* $(n_1, n_2, n_3, \cdots, n_t)$. The number of elements in each dimension is N_τ for $\tau = \{1, 2, \cdots, T\}$. That is, the number of elements in the first dimension is N_1 and so on. Then the 0-based index for the element with t-tuple $(n_1, n_2, n_3, \cdots, n_t)$ is

$$indx = n_t + N_t(n_{t-1} + N_{t-1}(n_{t-2} + N_{t-2}(n_{t-3} + N_{t-3}(\cdots + N_2 n_1))))$$

Since we are using 0-based indexing, n_τ must reside in the interval $\{0, 1, \cdots, N_\tau - 1\}$. As an example, let's element $a_{2,3,2}$ whose location in matrix A is $(1, 2, 1)$. Its 0-based index is $1 + 2 \times (2 + 3 \times 1) = 11$.

The individual addresses of the t-tuple are recovered with these sequential steps

1. $n_t = \frac{indx}{N_1 \times N_2 \cdots \times N_{\tau-1}}$,

2. $n_{t-1} = \frac{(indx \ - \ n_t \times (N_1 \times N_2 \cdots \times N_{\tau-1}))}{N_1 \times N_2 \cdots \times N_{\tau-2}}$,

3. $n_{t-2} = \frac{indx \ - \ n_t \times (N_1 \times N_2 \times \cdots \times N_{\tau-1}) \ - \ n_{t-1} \times (N_1 \times N_2 \cdots \times N_{\tau-2})}{N_1 \times N_2 \times \cdots \times N_{\tau-3}}$,

   $\vdots$

4. $n_1 = indx \ - \ n_t \times (N_1 \times N_2 \times \cdots \times N_{\tau-1}) \cdots - \ n_2 \times N_1$.

Correctly recovering these indexes is crucial for the execution of the model in the GPU. In C language, a division between integers returns an integer rounded towards zero. Hence,

the previous steps return the correct integer indexes. With this indexing system in hand, it is straightforward to implement the value-function-iteration approach to the default model.

4 Sovereign meets C++

Solving the model via value function iteration requires that for each grid in the state space of endowments and bonds ($\mathcal{Y} \times \mathcal{B}$) the sovereign solves her maximization problem. Let N_y and N_b be the grids of points containing the feasible sets of endowment and debt. The solution proceeds sequentially by first computing the value of defaulting, $V_d(y)$, the value of good standing, $V_r(y, b)$, and then the decision to default or not, $V(y, b) = max_{d \in \{0,1\}}\ (d\ V_d(y) + (1 - d)\ V_r(y, b))$. The resulting pseudocode is[3]

```
1  Initialization;
2  Define maxind = N_y × N_b;
3  while error > tol and iter < maxiter do
4      Ṽ = V ;
5      for index = 0; index < maxind; index++ do
6          Recover indexes: i_b = index/N_y and i_y = index − i_b × N_y ;
7          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)] ;
8          V_r(y(i_y), b(i_b)) = max_{b′} U(c) + βE[V(y′, b′)|y(i_y), b(i_b)] subject to budget
           constraint 2 ;
9          if V_r(y(i_y), b(i_b)) > V_d(y(i_y)) then
10             Repay: V(y(i_y), b(i_b)) = V_r(y(i_y), b(i_b))
11         else
12             Default: V(y(i_y), b(i_b)) = V_d(y(i_y))
13         end
14         Update debt price q(y(i_y), b(i_b)) = (1−prob(default))/(1+r*)
15     end
16     error = max(abs(Ṽ, V)) ;
17     Update value function: V = δ V + (1 − δ)Ṽ ;
18     iter++;
19 end
```

Algorithm 1: Pseudocode Default Model in C++

[3]The pseudocode may look familiar to implementations using Fortran or Matlab. The main difference is the indexing starting in 0 rather than the more usual 1.

5 Sovereign meets CUDA

The biggest challenge to migrate the code to CUDA is the presence of the "*for loop*" in the C++ pseudocode. A second obstacle comes from computing the optimal debt choice and default decision (line 9 in algorithm 1). Hence, most of the task ahead is to split the loop into pieces that can be sent separately to the GPU. For the sake of clarity, I skip the space consuming (but important) initialization of host and device vectors as well as the relevant pointers. The reader can consult the companion code for details.

A first step toward using CUDA was taken in the C++ solution. Note that the solution already uses arrays with 0-indexing order. Next, we need to make the code modular to send requests to the GPU. One alternative is to code the entire for-loop (lines 5 to 15 in Algorithm 1) into a single device module. In this way, the entire solution runs at once in the GPU. This option, however, can be very onerous because large vectors (containing value functions, prices, decision to default, and their values in the past iteration) must reside simultaneously in the device's global memory. Alternatively, we can split the solution into three blocks: 1) compute value of default V_d, 2) compute value of repayment V_r, and compute the decision to default, and 3) compute errors. Then send each of these blocks as separate requests to the GPU. This is precisely what Algorithm 2 does.

```
1  Initialization;
2  Define maxind = N_y × N_b;
3  while error > tol and iter < maxiter do
4  |   Ṽ = V ;
5  |   thrust::for_each(begin,end,compute_value_default(.));
6  |   thrust::for_each(begin,end,findmax(.));
7  |   error = thrust::transform_reduce(
   |   make_zip_iterator(make_tuple(valueold.begin(), value.begin())),
   |   make_zip_iterator(make_tuple(valueold.end(), value.end())), myMinus(), 0.0,
   |   maximum<double>()) ;
8  |   iter++;
9  |   Update value function: V = δ V + (1 − δ)Ṽ ;
10 end
```

Algorithm 2: Pseudocode Default Model in CUDA

There are several features in Algorithm 2 worth discussing. First, the code uses heavily Thrust, which is "a C++ template library for CUDA." Thrust has the advantage of hiding onerous and complex lower level programming tasks such as memory management. As a

N_b	C++	CUDA
300	12s	3s
500	36s	4s
1000	2m38s	14s
5000	NA	4m26s

Table 1: Computing Times

consequence, programming using Thrust achieves a greater level of abstraction than using CUDA alone. Indeed, the simplicity of Algorithm 2 illustrates the power of combining CUDA and Thrust. Second, the command for_each in line 5 instructs the GPU to compute the value of defaulting for each of the elements in our grid (begin,end). This value of defaulting is implemented in the structure compute_value_default (see list 1 in the appendix). Third, the heavy lifting is done by the for_each command in line 6 (list 2). Among other things, this step computes for each grip point the value of repayment, the decision to default, and updates the price of debt. Fourth, the directive transform_reduce uses the functor myMinus (see list 3 in the appendix) to compute the value function error in the most recent iteration. Finally, it is worth emphasizing that the algorithm is highly scalable. Indeed, the same algorithm has been successfully applied to the bastly more complicated default model in Gordon and Guerron-Quintana (2016b)

In Algorithms 1 and 2, δ is a relaxation parameter to improve convergence. For a grid with 300 points for debt and 21 points for endowment and a tolerance on the value function errors of $1e-6$, it takes 12 seconds for C++ (using the GNU g++ compiler version 5.2.0) to solve the model in an iMac 3.5 GHz Intel i7 with 16 GB of RAM. Solving the model compiled with the CUDA compiler (version 7.5.19) using a NVIDIA GeForce GTX 780 card in the same iMac takes around 3 seconds. This is a speedup of 4 times relative to the C++ version. The superior performance of CUDA is even clearer as we increase the size of the debt grid (Table 1). Even with the very fine grid using 5,000 points for debt, CUDA requires only 4 and a half minutes.

References

E. Aldrich,J. Fernandez-Villaverde, R. Gallant, and J. Rubio-Ramirez. Tapping the supercomputer under your desk. *Journal of Economic Dynamics and Control*, 35386–393, 2011.

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

B. Aruoba and J. Fernandez-Villaverde. A comparison of programming languages. *Journal of Economic Dynamics and Control*, 58265–273, 2015.

J. Eaton and M. Gersovitz. Debt with potential repudiation: Theoretical and empirical analysis. *The Review of Economic Studies*, 48(2):289–309, 1981.

G. Gordon and P. Guerron-Quintana. Dynamics of investment, debt, and default. Working Paper, Federal Reserve Bank of Philadelphia, 2016a

G. Gordon and P. Guerron-Quintana. Regional default and migration. Working Paper, Federal Reserve Bank of Philadelphia, 2016b

J. Hatchondo, L. Martinez, and H. Sapriza. Quantitative properties of sovereign default models. *Review of Economics Dynamics*, 13919–933, 2010.

D. Kirk and W. Hwu. Programming Massively Parellel Processors. *Morgan Kaufmann, MA.* Second Edition, 2013.

6 Appendix

Description of structures used in Algorithm 2. The full version is available in the companion code.

Listing 1: **Compute value of default**

```
struct compute_value_default{

__host__ __device__
compute_value_default(){P = P_ptr; Y = Y_ptr; vbad = vbad_ptr;
wdef = wdef_ptr; value = value_ptr ;}

__host__ __device__
void operator()(int index){
int i_b = index/(ny) ;
int i_y = index - ny * i_b ;
double sumdef, consdef ;
```

```
sumdef = 0.0;
consdef = pow(exp(Y[i_y])*(1 - ppi), (1 - rrisk)) / (1 - rrisk);
for (int ttx = 0; ttx < ny; ttx++) sumdef += P[i_y + ny * ttx] *
((1-ttheta)*vbad[ttx]+ttheta*value[ttx+ny*(nb-1)]) ;
def[i_y] = consdef + bbeta*sumdef;
}
};
```

Listing 2: **Compute whether to default or not**

```
struct findmax{

__host__ __device__
findmax(){B = B_ptr; Y = Y_ptr ; qrB = qrB_ptr; Y = Y_ptr;
EV = EV_ptr; value = value_ptr ; wdef = wdef_ptr;
decision = decision_ptr ; vbad = vbad_ptr; P = P_ptr,
qprupd = qprupd_ptr ;};

__host__ __device__
void operator()(int index){
int i_b = index / (ny);
int i_y = index - i_b*ny;
double vlar = 0.0, vfirst = 0.0, sumdef = 0.0, sumbad = 0.0 ;
double budget, consdef ;
int left_ind = 0 ;
int right_ind = nb ;
int largest = left_ind, first = left_ind ;

budget = B[i_b] + exp(Y[i_y]) ;

while (first < right_ind) {

(budget > qrB[i_y + ny*largest]) ? vlar =
pow(budget - qrB[i_y + ny*largest], (1 - rrisk)) / (1 - rrisk) +
   bbeta * EV[i_y + ny*largest] : vlar = -10000000.0;
(budget > qrB[i_y + ny*first]) ? vfirst = pow(budget - qrB[i_y + ny*
   first], (1 - rrisk)) / (1 - rrisk) + bbeta * EV[i_y + ny*first] :
    vfirst = -10000000.0;
```

```
if (vlar < vfirst)
largest = first;
first++;
}

if (vlar > wdef[i_y]){
value[index] = vlar ;
decision[index] = 0 ;
}
else{
value[index] = wdef[i_y] ;
decision[index] = 1 ;}

// Update prob of default
sumdef = 0.0;
for (int dfx = 0; dfx < ny; dfx++) sumdef += P[i_y+ny*dfx] *
   decision[dfx+ny*i_b];
// Update debt price
qprupd[index] = (1.0 - sumdef) / (1 + rstar);

sumbad = 0.0 ;
consdef = pow(exp(Y[i_y])*(1-ppi),(1-rrisk))/(1-rrisk) ;
for (int ttx = 0; ttx < ny; ttx++) sumbad += P[i_y+ny*ttx]*((1-
   ttheta)*vbad[ttx]+ttheta*value[ttx+ny*(nb-1)]) ;
vbad[index] = consdef + sumbad*bbeta ;
}
};
```

Listing 3: **Compute difference between two vectors**

```
struct myMinus {
        template <typename Tuple>
        __host__ __device__
        double operator()(Tuple t)
        {
        return abs(get<0>(t)-get<1>(t));
        }
};
```