GPU-Accelerated Value Function Iteration in Julia: Faster Macroeconomic Modeling

Giovanni Ballarin
@giob1994

University of Konstanz - University of Rome Tor Vergata

2018

Disclaimer

I am not a programmer (unfortunately for this talk)
and my code might be terribly ugly...

Sorry in advance if what follows causes you discomfort.

Github Repository

All the code used is in Jupyter notebooks available on GitHub:

`https://github.com/giob1994/OpenCL-VFI-in-Julia`

Table of Contents

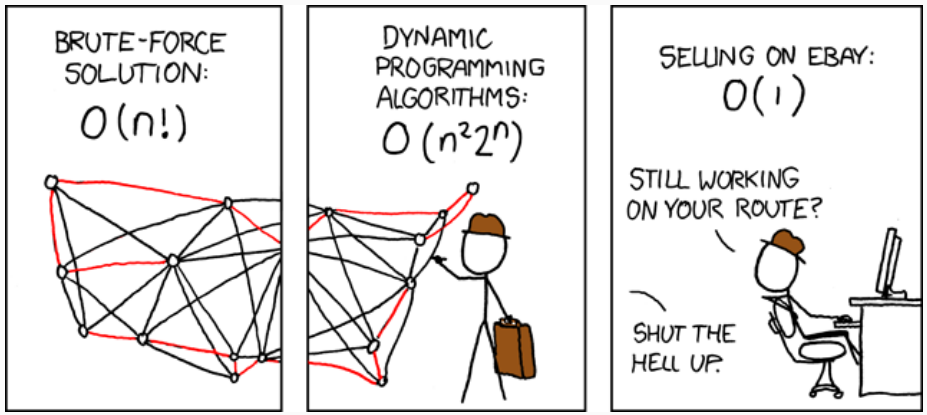
Table of Contents

Dynamic Programming Basics

Value **F**unction **I**teration (**VFI**) - also known as Value Iteration - is a well-know solution method for *dynamic programming* problems.

Dynamic Programming (**DP**) is a theoretical and practical framework for mathematical optimization, and is very commonly used in many fields:

- Control theory
- Programming
- **Economics**



© xkcd (3d.xkcd.com/399/)

Value Function Iteration

Value Function Iteration is one way of solving numerically the Bellman equation: notably, the *Contraction Mapping Theorem* ensures that VFI converges to the correct solution for *any* initial guess.

The generic problem is

$$V^n(S) = \max_{\{\text{actions}\}} \left\{ u(\text{actions}, S) + \beta V^{n-1}(S') \right\}$$

Define the operator $\mathbf{T}$

$$V^n(\cdot) = \mathbf{T} V^{n-1}(\cdot)$$

Value Function Iteration

VFI Algorithm

1. Start with an initial guess $V_0(\cdot)$
2. Compute $V^1(\cdot) = \mathbf{T}\, V^0(\cdot)$
3. Iterate $V^n(\cdot) = \mathbf{T}\, V^{n-1}(\cdot)$
4. Stop when $||V^n(\cdot) - V^{n-1}(\cdot)|| < \epsilon$

Value Function Iteration

Note:

- V cannot be handled directly!
- V is a *function*, an thus infinitely dimensional...
- **Solution**:
  Discretize the domain, and approximate V using a finite set of points.

Value Function Iteration

The Good:

- VFI solves practically all DP problems
- VFI always converges (if a solution exists)
- VFI is easy to grasp conceptually
- VFI is also straightforward to implement

The Meh:

- VFI gets a bit tricky with the addition of a stochastic component

The Bad:

- VFI suffers from the **curse of dimensionality**

Table of Contents

A Simple Economic Problem

- We now want to apply VFI to a simple economic model
- Start with the **RBC** (**R**eal **B**usiness **C**ycle) model, which is a benchmark macroeconomic model
- The RBC is very basic, but requires* the use on numerical tools to get to a solution!

Features of the RBC:

1. Aggregate economy
2. Agents (consumers) and firms (production)
3. Deterministic dynamics

   (But we will later add a random component...)

A Simple Economic Problem

$$\mathbb{E}\sum_{t=0}^{+\infty}\left\{\beta^t\frac{c_t^{1-\sigma}}{1-\sigma}\right\}$$

subject to:

$$\begin{aligned} c_t + i_t &= y_t \\ k_{t+1} &= (1-\delta)k_t + i_t \\ y_t &= k_t^{\alpha} \end{aligned}$$

- c_t: consumption
- k_t: capital
- i_t: investment
- y_t: production
- β: time discount
- σ: elasticity
- δ: depreciation

Bellman Equation:

$$V(k) = \max_{k'}\left\{\frac{c_t^{1-\sigma}}{1-\sigma} + \beta V(k')\right\} \quad \text{s.t.} \quad k' = k^{\alpha} + (1-\delta)k - c$$

VFI for the RBC model

1. Fix a discrete grid for capital, K: denote the number of points in this grid by N_k, set $i = 0$ and set $\varepsilon > 0$ to be a small tolerance level (for convergence)
2. Make an initial guess for V^0 (a good initial guess will speed up convergence)
3. **while** $\| V^{i+1} - V^i \| > \varepsilon$
4. **for** each $k \in K$ **do**
5. Compute

$$W(k, k') = u(k^\alpha + (1 - \delta)k - k') + \beta V^i(k')$$

for each $k' \in K$.

6. Set $V^{i+1}(k) = \max_{k'} \{W(k, k')\}$.
7. **end for**
8. **end while**

VFI for the RBC model

Julia code:

```
it = 0;
while ( sum(abs.(V_prev — V_)) > prec_ && it < maxiter_ )
    V_prev = copy(V_);
    for (i, k) in enumerate(Kgrid)
       V_max = —Inf;
        for (j, k1) in enumerate(Kgrid)
            c = (k^alpha + (1—delta)*k — k1);
            if c > 0
                V_max = max(V_max, (log(c) + beta*V_prev[j]));
            end
        end
        V_[i] = V_max;
    end
    it += 1;
 end
```

The curse of dimensionality

- There are some issues with the previous code...
- It's not optimized at all
- Vectorization is key for performance

But there is a fundamental issue...

- As we increase the size of the grid, the number of points to evaluate increases... ***quadratically***
- This can be mitigated by vectorized ops and pre-computation... but not by much
- Unless we tap into vector CPU instructions (e.g. AVX2), the speedup margin falls flat very early...

Table of Contents

GPGPU for Economists

- GPUs can not replace CPUs in ***all*** workloads
- Superior performance in embarrassingly parallel mathematical tasks
- Economists should be very interested in GPGPU...
- **Incredible speedups!**
  Depending on application, up to 1000× faster exec.
- **Wide applicability** to many economical problems
  (e.g. VFI, Monte Carlo, repeated games...)

Example:

"*With 65,536 capital grid points and counting the memory allocation, the GPU is roughly* ***509 times faster*** [at VFI with binary search] ..."

Journal of Economic Dynamics & Control

Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors

Aldrich, Eric M., Fernandez-Villaverde, Jesus, Gallant, A. Ronald & Rubio-Ramirez, Juan F. "**Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors.**", *Journal of Economic Dynamics and Control* 35, no. 3 (2011): 386-393.

GPU VFI

Amdahl's Law: if your algorithm is *not parallel enough*, GPUs will not help *at all*...

Is VFI compatible with GPUs parallelism?

- **Yes!**
- VFI is especially well suited for GPGPU...

GPU VFI

1. Fix a discrete grid for capital, K: denote the number of points in this grid by N_k, set $i = 0$ and set $\varepsilon > 0$ to be a small tolerance level (for convergence)
2. Make an initial guess for V^0 (a good initial guess will speed convergence)
3. **while** $\| V^{i+1} - V^i \| > \varepsilon$
4. **for** each $k \in K$ **do**
5. Compute

$$W(k, k') = u(k^\alpha + (1 - \delta)k - k') + \beta V^i(k')$$

for each $k' \in K$.

6. Set $V^{i+1}(k') = \max_{k'} \{W(k, k')\}$.
7. **end for**
8. **end while**

GPU VFI

Amdahl's Law: if your algorithm is *not parallel enough*, GPUs will not help *at all*...

Is VFI compatible with GPU parallelism?

- **Yes!**
- VFI is especially suited for GPGPU...
- VFI operates on **each** element of the discrete capital grid **separately**: the search for the optimal $V^{i+1}(k) \ \forall \, k$ can be done in parallel!

VFI with OpenCL in Julia

- **Julia** makes makes it easy to use the GPU
- Many packages from the *JuliaGPU* group!
- Interfaces for CUDA *and* OpenCL

OpenCL.jl

- Write an OpenCL kernel for VFI
- Each thread solves for a single $k \in K$

OpenCL Kernel

```
__kernel void opencl_vfi( __global float *V0, __global float *V,
                          __global const float *grid, const int grid_size) {
    int gid      = get_global_id(0);
    float alpha  = 0.5f;
    float beta   = 0.7f;
    float grid_p = grid[gid];
    float V_tmp  = -INFINITY;
    float u_arg;
    float V_new;
    for(int i = 0; i <= grid_size; i++) {
        u_arg = pow(grid_p, alpha) - grid[i];
        if (u_arg > 0) {
            V_new = log(u_arg) + beta*V0[i];
            if (V_tmp < V_new) { V_tmp = V_new; }
        }
    }
    V[gid] = V_tmp;
}
```

Julia code - Function

```
using OpenCL
# Create the grid
Kgrid = Array{Float32}(collect(linspace(0.1, 100, 1000)));
# Set up the OpenCL device
device, ctx, queue = cl.create_compute_context();
# Host arrays
V0 = Array{Float32}(zeros(size(grid_)));
V  = Array{Float32}(zeros(size(grid_)));
# Device (GPU) buffers
grid_buff = cl.Buffer(Float32, ctx, (:r, :copy), hostbuf = Kgrid);
V0_buff   = cl.Buffer(Float32, ctx, (:r, :copy), hostbuf = V0);
V_buff    = cl.Buffer(Float32, ctx, :w, length(V));
# Compiling the kernel
p = cl.Program(ctx, source = vfi_cl) |> cl.build!
krn = cl.Kernel(p, "opencl_vfi")

# [! ... Run the OpenCL kernel ... !]
```

Julia code

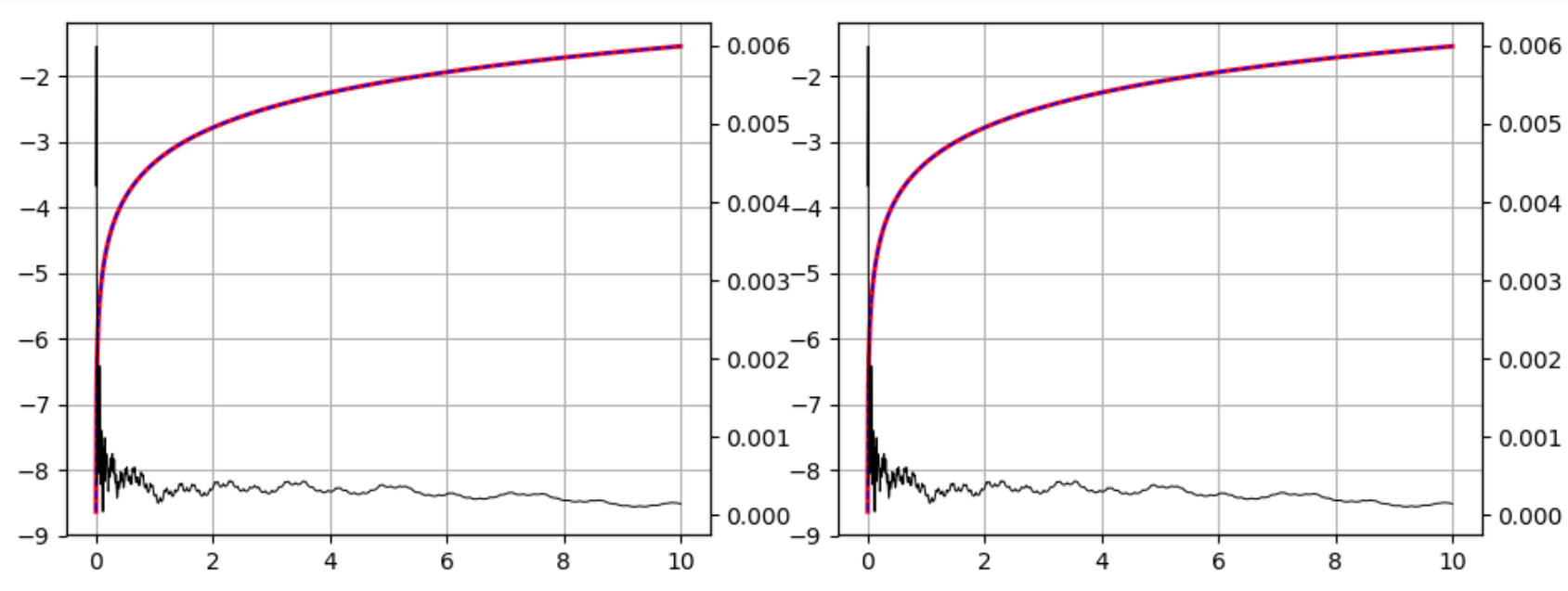
```
# [! ... Run the OpenCL kernel ... !]

# Initialize local Value Function vectors
V0 = ones(size(V0));
V1 = zeros(size(V0));
it = 0;
while ( norm(V0 — V) > prec_ && it < maxiter_)
    copy!(V0, V);
    # Running the OpenCL kernel
    krn[queue, size(grid_)](V0_buff, V_buff, grid_buff, 1000)
    # Reading the results
    V  = cl.read(queue, V_buff);
    V1 = cl.read(queue, V0_buff);
    # Preparing for next iteration
    cl.copy!(queue, V0_buff, V_buff);
    it += 1;
end
```

OpenCL results

Does it work?

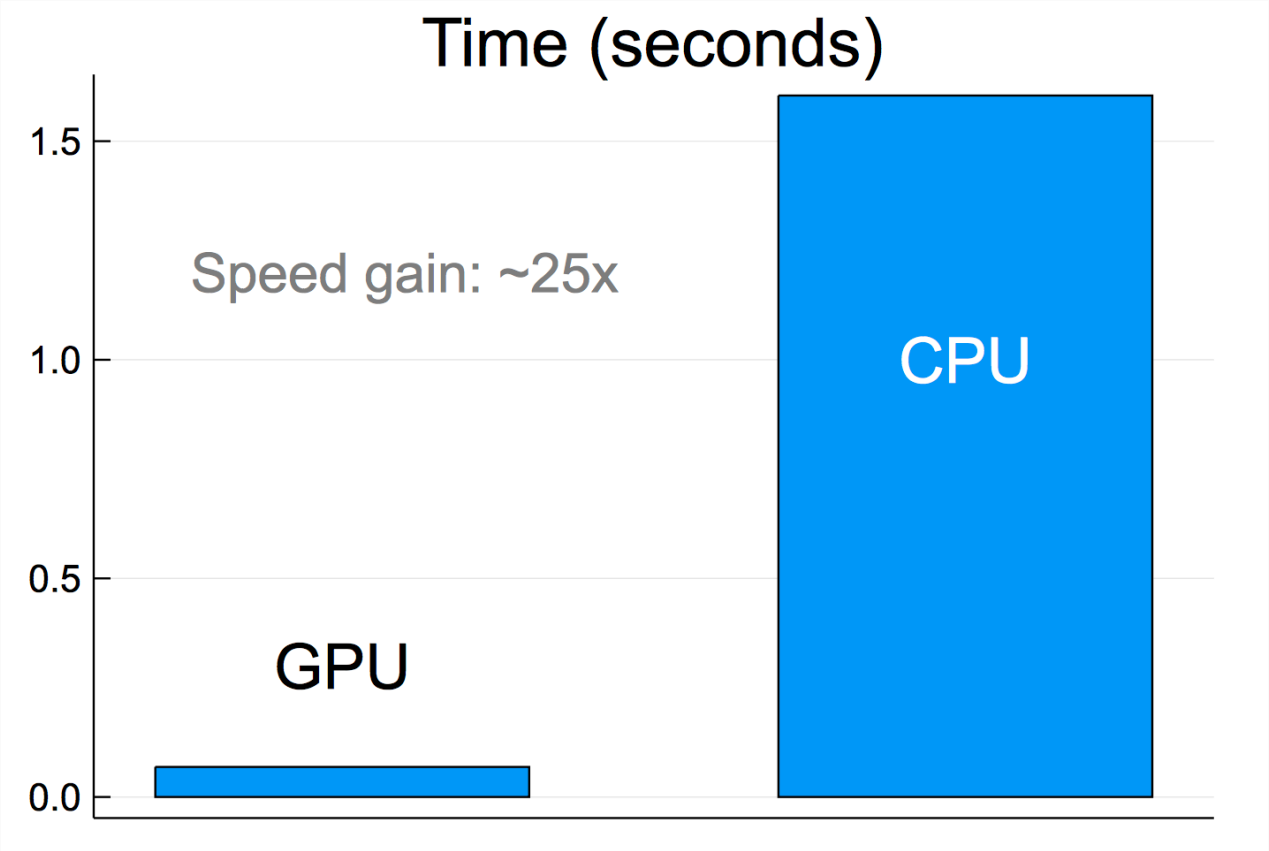
*Well **yes**, otherwise it would be pretty pointless...*



OpenCL results

Is it faster?

*Well **yes**, otherwise it would be pretty pointless...*



OpenCL results

Is it NICE?

Not really, *unless you really enjoy*

1. *writing OpenCL code*
2. *debugging OpenCL code*
3. *debugging Julia code to execute OpenCL kernels*
4. *debugging OpenCL code...*

Julia must have a better way, right?

Table of Contents

GPUArrays.jl

GPUArrays.jl[1] is an awesome package that allows us to execute GPU code and kernels by using *pure Julia*.

CLArrays.jl offers a concrete implementation for OpenCL, but **CuArrays.jl** (CUDA) is also available.

By using this package, we do not need to write separate OpenCL code: everything can be done in Julia!

[1] `https://github.com/JuliaGPU/GPUArrays.jl`

RBC with uncertainty

We now consider a model with a stochastic component: productivity (of capital) now depends on a **random process**, z.

$$V(k) = \max_{k'} \left\{ \frac{c_t^{1-\sigma}}{1-\sigma} + \beta \mathbb{E}[V(k')] \right\} \quad \text{s.t.} \quad k' = z\, k^{\alpha} + (1-\delta)k - c$$

where

$$\mathbb{E}\left[V_{z_i}(k)\right] = \sum_{j=1}^{N_z} \left[P_{i,j} \cdot V_{x_j}(k)\right]$$

and P is a matrix of transition probabilities.

RBC with uncertainty

- Random process z is a Markov process determined by P.
- Let N_z be the number of states for z...
- The agent must weight the chances of *all* possible futures.
- This new model requires computing N_z value functions!

Even with a small complication, it necessary to **multiply** our previous solution efforts by N_z times!

If z is an approximation of a continuous random process (see Tauchen), then to have reasonable precision then computational complexity of the VFI might get immediately too high.

CPU

- Create matrices to store value functions:

```
V0 = Array{Float32}(ones(SIZE_KGRID, SIZE_ZGRID));
V  = Array{Float32}(ones(SIZE_KGRID, SIZE_ZGRID));
```

- Repeat for every capital grid point and random process value:

```
# For every point on the capital grid:
for i in 1:SIZE_KGRID
    # For every state of productivity:
    for z_ind in 1:SIZE_ZGRID
        # [ ... ]
    end
end
```

CPU

- Compute consumption for all possible k':

```
tmp = Zgrid[z_ind] * Kgrid[i]^alpha +
                              (1—delta)*Kgrid[i] .— Kgrid;
```

- For every possible value of consumption, find the maximum value and its position on the capital grid:

```
for (j, c_j) in enumerate(tmp)
   # If consumption is positive:
   if point_j > 0
      tmp_comp = (c_j^(1—sigma)—1)/(1—sigma) +
                              beta*dot(P[z_ind, :], V0[j, :]);
      # [ ... Find the max of the Value Function ... ]
   end
end
```

GPU with GPUArrays!

- Create matrices to store value functions on the GPU:

```
V      = ones(CLArray{Float32}, SIZE_KGRID, SIZE_ZGRID);
grid   = CLArray(Kgrid);
z      = CLArray(Zgrid);
P      = CLArray(P);     # transition matrix
```

- Write the kernel...

```
gpu_call(Kgrid, (Kgrid, V, policy, Zgrid, P, Float32(alpha),
            Float32(beta), Float32(delta), Float32(sigma),
            UInt32(SIZE_KGRID), UInt32(SIZE_ZGRID)))
         do state, Kgrid, V, Zgrid, P, alpha, beta, delta,
            sigma, SIZE_KGRID, SIZE_ZGRID
     # [ ... Kernel ... ]
end
```

RBC with uncertainty

GPU with GPUArrays!

- Each kernel executes for one value of the capital grid:

```
idx = @linearidx grid
# Kgrid[idx] is the grid point that we solve for...
```

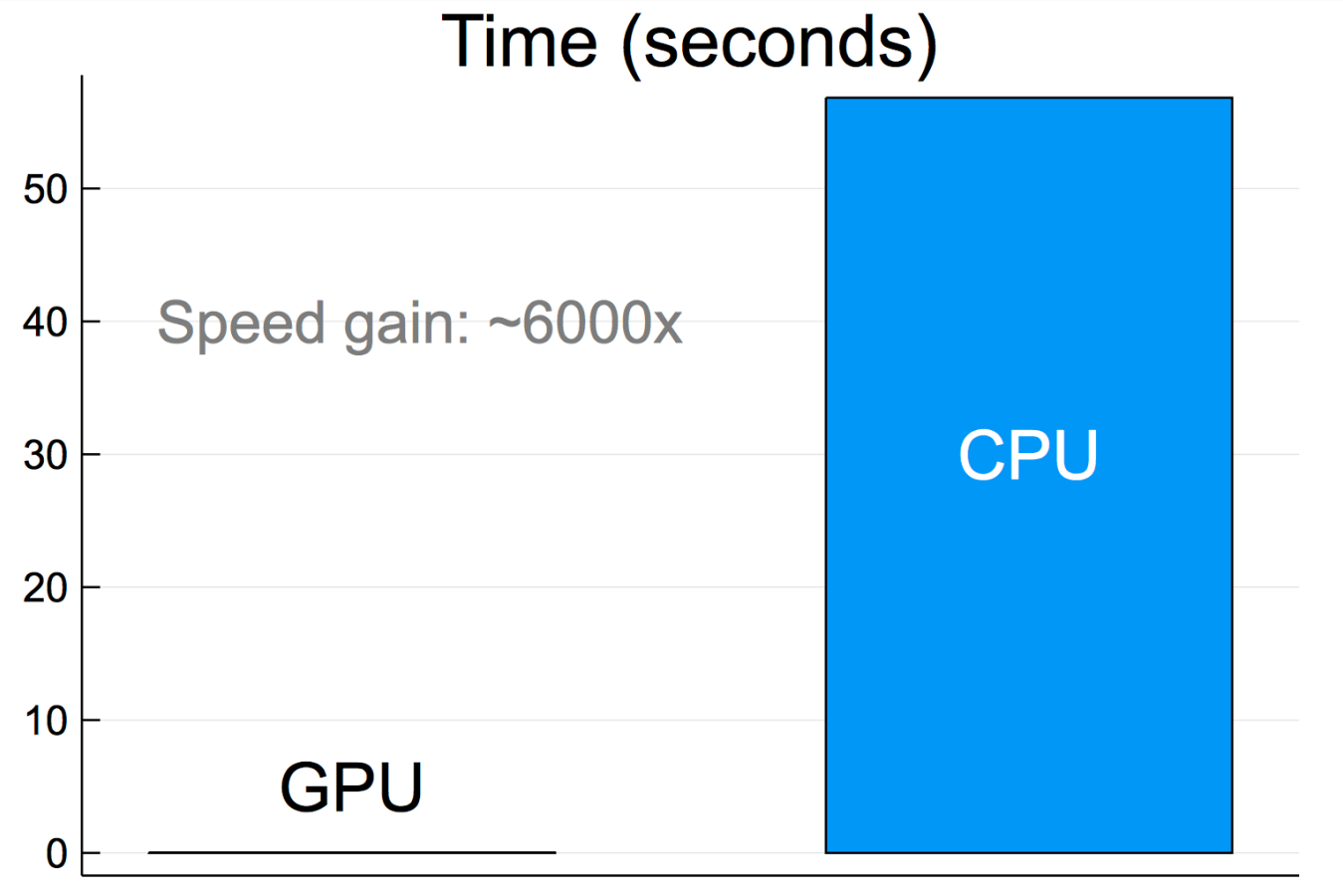
- Write the kernel...

  *The GPU kernel has the **same*** **code** as the CPU function!*

GPUArrays.jl allows us to re-use the same Julia code of the "standard" VFI function to seamlessly code OpenCL kernels.

OpenCL results

Is it *still* faster?



OLG Models very very fast...

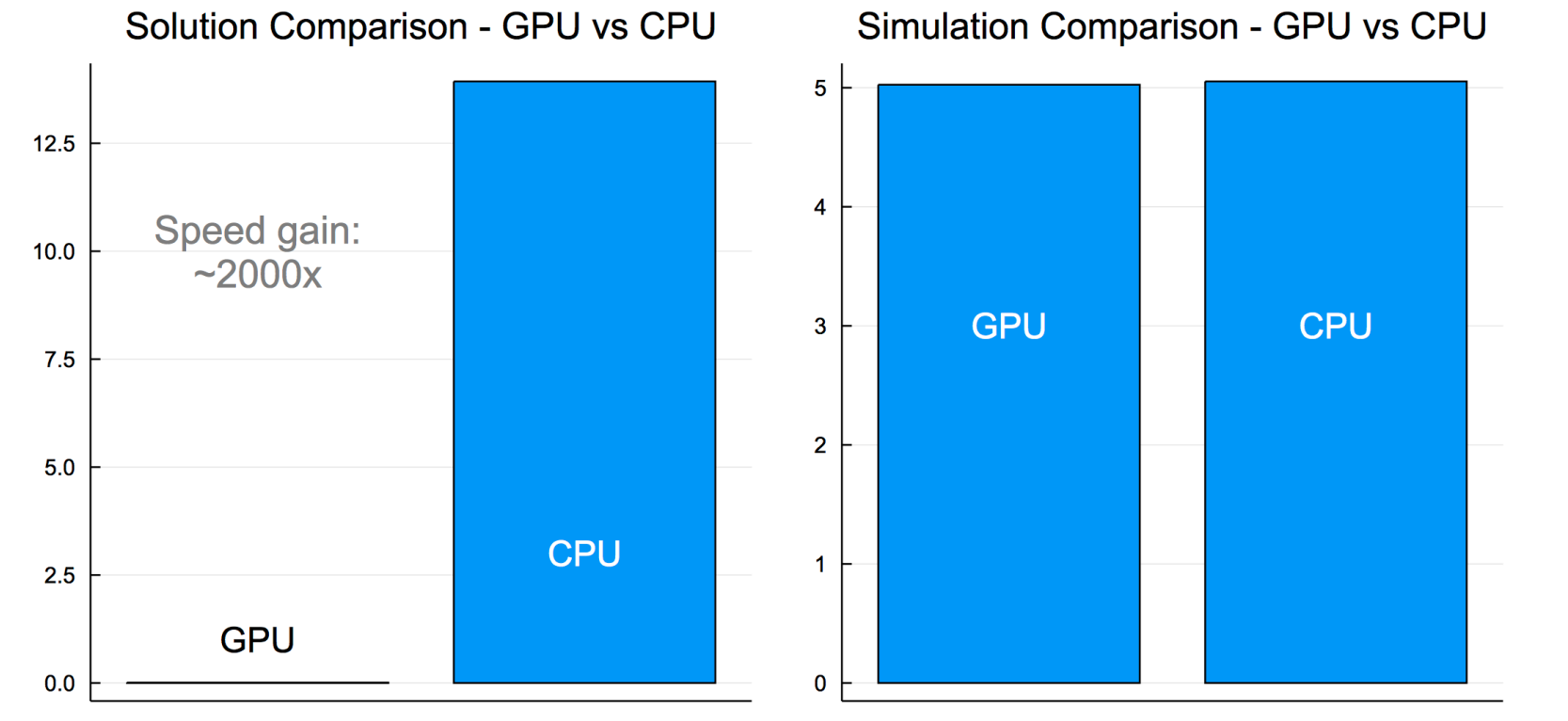
- OLG models have agents with lifetimes, so there are T value functions to solve!
- OLG models are usually solved by *backward iteration*: starting from the last period back to the first
- If we throw in capital, human capital, ability etc... Might have to solve for **millions** of points!
- *Convergence* is also required: solve the model multiple times...
- Takes **hours**

OLG Models very very fast...

Using the GPU:

- Since the kernel is the same, and the cost of compilation & initialization is fixed
- Assumption: everything fits on the GPU
- But for "complex" OLG model, GPUArrays.jl can provide 150× performance*
- From hours to *minutes*

OLG Models very very fast...



GPUArrays results

Does it work?

- Relative errors of the value function are small, even close to 0!

Is it faster?

- With GPUArrays, solutions are around **3 orders of magnitude** faster!

Is it nice?

- Compared to "bare" OpenCL, ***Julia + GPUArrays is breeze!***

Conclusion

Conclusion

1. If parallelism is possible, take advantage of it!
2. GPGPU brings big returns to efforts (if you're lucky)
3. In Julia, coding kernels is much simpler than getting hand dirty with actual GPU kernel code
4. Economists, be **bold** and experiment with GPGPU!

Acknowledgment

Simon Danisch:

CLArrays.jl & Transpiler.jl make all of this possible!