Model Standardization, Efficient Compilation Schemes and Solution Algorithms

Pablo Winant

IMF, June 28

In [2]:
```python
from dolo import *
model = yaml_import("../code/rbc_fga.yaml")
model
```

Out[2]: Compiled model : rbc_fga

	Equations	Residuals
transition		
	$z_t = \rho z_{t-1} + zbar(-\rho + 1) + e_{z,t}$	0.0
	$k_t = i_{t-1} + k_{t-1}(-\delta + 1)$	0.0
arbitrage		
	$1 = \beta\left(\frac{c_t}{c_{t+1}}\right)^{\sigma}(-\delta + rk_{t+1} + 1)$	-1.94289029309e-16
	$-\chi c_t^{\sigma} n_t^{\eta} + w_t = 0$	-2.22044604925e-16
auxiliary		
	$c_t = -i_t + k_t^{\alpha} n_t^{-\alpha+1} z_t$	0.0
	$rk_t = \alpha z_t\left(\frac{n_t}{k_t}\right)^{-\alpha+1}$	-2.77555756156e-17
	$w_t = z_t\left(\frac{k_t}{n_t}\right)^{\alpha}(-\alpha + 1)$	0.0

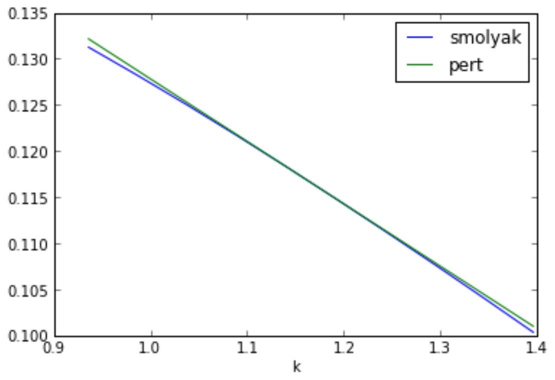
Solving the model

In [3]:
```
dr_pert = approximate_controls(model)
%time dr_global = global_solve(model, smolyak_order=3)
```

```
CPU times: user 304 ms, sys: 2.35 ms, total: 306 ms
Wall time: 313 ms
```

In [4]:
```
plot_decision_rule(model, dr_global, 'k', plot_controls='i', label='smolyak')
plot_decision_rule(model, dr_pert, 'k', plot_controls='i', bounds=dr_global.bounds[:,1]
, label='pert')
legend()
```

Out[4]:
```
<matplotlib.legend.Legend at 0x4275950>
```



Our goals:

- define model independently from the solution algorithm
- implement generic solution
- as efficiently as possible
- provide elementary bricks to implement new ones
- without low-level code if possible
- open/free software: http://albop.github.io/dolo/

This presentation

- model standardization
- present compilation techniques to make model evaluation fast
  - maybe single computers can still do more
  - use vectorization before parallelisation
  - avoid memory problems
- compare cpu vs gpu implementation on simple a time-iteration algoritm

Model standardization

- Each software represents equivalent models in different ways
    - Dynare, IRIS, RISE,
- No de facto standard for global models
    - JBendge: a model file
    - compecon: an API standard
- Advantages of choosing a common representation:
    - compare solutions more easily / use starting values
    - reuse code
- Proposition based on serialization format YAML
- Problem: each algorithm uses different information

An "fga" model

- vector of variables:
  - states: s_t
  - controls: x_t
  - auxilaries: y_t

- model given by $f, g, a, \underline{b}, \overline{b}$
  - Controlled process:

$$s_t = g(s_{t-1}, x_{t-1}, y_{t-1}, \epsilon_t)$$

  - Optimality conditions:

$$f(s_t, x_t, y_t, s_{t+1}, x_{t+1}, y_{t+1}) \perp \underline{(}b) \leq x_t \leq \overline{b}$$

  - Auxiliary variables:

$$y_t = a(s_t, x_t)$$

In [16]:
```
!head -27 ../code/rbc_fga.yaml
```

```
model_type: fga

declarations:

   states:  [z, k]
   controls: [i, n]
   auxiliary: [c, rk, w]
   shocks: [e_z]

   parameters: [beta, sigma, eta, chi, delta, alpha, rho, zbar ]


equations:

   arbitrage:
      - 1 = beta*(c/c(1))^(sigma)*(1-delta+rk(1))   | 0 <= i <= inf
      - w - chi*n^eta*c^sigma                       | 0 <= n <= inf

   transition:
      - z = (1-rho)*zbar + rho*z(-1) + e_z
      - k = (1-delta)*k(-1) + i(-1)

   auxiliary:
      - c = z*k^alpha*n^(1-alpha) - i
      - rk = alpha*z*(n/k)^(1-alpha)
      - w = (1-alpha)*z*(k/n)^(alpha)
```

Generic structure

```
 - declarations (by type)
 - equations (by type)
 - calibration
```

Expected content is defined- in a `recipe` file

Other types of models (1)

fgh model

$s_t = g(s_{t-1}, x_{t-1}, \epsilon_t)$
$f(s_t, x_t, z_t)$
$z_t = h(s_t, x_t)$

Other types of models (3)

vfi model

$$s_t = g(s_{t-1}, x_{t-1}, \epsilon_t)$$

$$V_t = \max_{x_t} U(s_t, x_t) + \beta V_{t+1}$$

Other types of models (4)

dynare model (no explicit states)

$E_t\left[f\left(y_{t+1}, y_t, y_{t-1}, \epsilon_t\right)\right]$

Many, many other possibilities

- Kumhof, Ranciere and Winant (2013) : fga model + endogenous distribution of shocks
- Coeurdacier, Rey and Winant (2013) : fga model + discrete markov states

What makes the solution fast ?

Solution algorithm:

- initialization: initial guess $x0[:,:]$ for the controls on the grid $s[:,:]$
- step k+1:
  - choose $x[:,:]$ for the controls on $s[:,:]$
    - compute states tomorrow $S[:,:] = g(s[:,:], x[:,:], e_m)$ for each shock e_m
      - interpolate $x_k[:,:]$ on $S[:,:]$ to get future controls
      - compute residuals $R_m = f(s, x, S, X)$
    - compute residuals $R = \sum_m w_m R_m$ by integrating the shocks
  - find optimal value x_{k+1} in 1/ using a nonlinear solver
- iterate until $||x_{k+1} - x_k||_{\infty} < 1e-8$

Ingredients:

- fast evaluation of f and g
- interpolation routines
- nonlinear solver for sparse systems

Solution algorithm:

- initialization: initial guess $x0[:,:]$ for the controls on the grid $s[:,:]$
- step k+1:
    - choose $x[:,:]$ for the controls on $s[:,:]$
        - compute states tomorrow $S[:,:] = g(s[:,:], x[:,:], e_m)$ for each shock e_m
            - interpolate $x_k[:,:]$ on $S[:,:]$ to get future controls
            - compute residuals $R_m = f(s, x, S, X)$
        - compute residuals $R = \sum_m w_m R_m$ by integrating the shocks
    - find optimal value x_{k+1} in 1/ using a nonlinear solver
- iterate until $||x_{k+1} - x_k||_\infty < 1e-8$

Remarks:

- the algorithm is perhaps not embarassingly parallel
- the vectorized algorithm has some costs:
    - "overoptimization" of some points
    - storage of intermediary results in main memory
- huge benefits

How to make model evaluation fast ?

problem:

- N: big integer
- s: $N \times 2$ matrix (states today)
- x: $N \times 2$ matrix (controls today)
- S: $N \times 2$ matrix (states today)
- X: $N \times 2$ matrix (controls today)
- res: $N \times 2$ matrix (residuals today)

Equilibrium conditions: compute $res = f(s, x, S, X)$

- How to compute model equations very fast ?
  - apply symbolic optimizations (rewrite expressions, compute common terms only once)
  - maybe you need low level programming (C, Fortran)
  - use the hardware efficiently:
    - vectorized
    - multicore
    - gpu

Interpreted vs. compiled languages

- Version in C: `C void fun(int N, double* a, double* b, double* c) {     for(int i=0; i<N; i++) {         c[i] = (a[i]*b[i] + 1)*b[i]     } }`
- Version in Python: `def fun(a,b,c):         for i in range(s.shape[0]):             res[i] = (a[i]*b[i] + 1)*b[i]         return res`

Are interpreted language slow ?

Informal conjecture:

- interpreted languages (Matlab, Python, R) are always slower than compiled languages

Informal lemma:

- it is impossible to beat naive C/Fortran code

Reformulation of Informal lemma:

- it is impossible to beat automatically optimized C code

Are interpreted languages slow ?

- Why are there slow ?
  - a lot of type-checking
  - non trivial types: slow boxing/unboxing of data
- Remedies:
  - avoid loops
  - vectorize code and call low-level libraries for intensive computations (Matlab, Numpy, ...)

High level vectorization

```python
def f(a, b, c):
    res = (a*b + 1)*b
```

- The code operates on arrays element by element
- Maximum efficiency for each element
- Slower than C because of the intermediary terms stored in main memory
- Possible solution: lazy evaluation
  - easy at run time (implemented in Theano)
  - complicated at compile time with templates

Two game-changers:

- research/implementation on Just-In-Time compilation
- Low Level Virtual Machine (LLVM): make it easy to write your own compiler

Just in time compilation

- pypy:
  - tracing compiler, recovers type information from running the code, then optimize. Can run almost any python code.
- numba:
  - observe types of input, recover type information, translate and compile `python @autojit def f(a,b,c): for i in range(s.shape[0]):      res[i] = (a[i]*b[i] + 1)*b[i]`
  - type of arguments is determined when function is called
  - other types are deduced by type inference
  - emits code equivalent to C !
- julia:
  - a scientific language designed so that type inference works as well as possible

Conclusion:

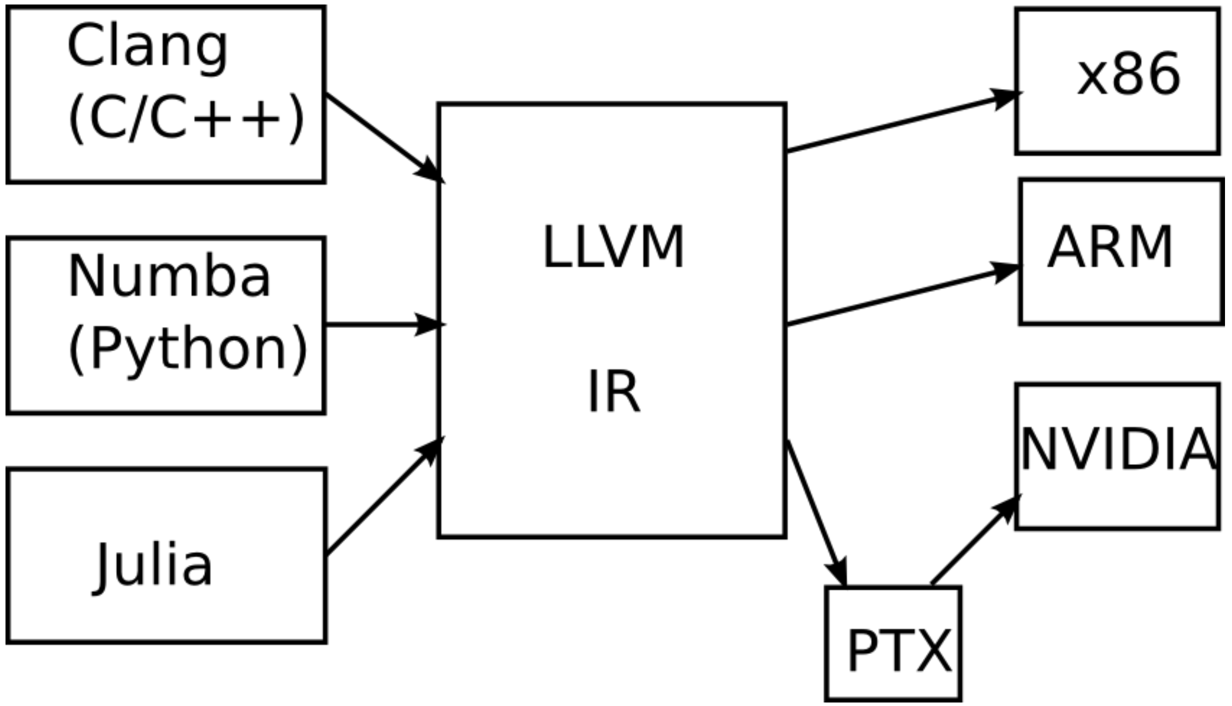
If JIT works well, no gain of using C unless you program *advanced* features yourself (resource access, cache management)

LLVM

- Design your own compiler, using a high level assembly language (machine independent)
- Apply low level optimizations, keeping high level informations
- Implement your own device driver for LLVM
- Easy for JIT compilation

In [7]:
```python
Image('../code/llvm.png', width=400)
```

Out[7]:



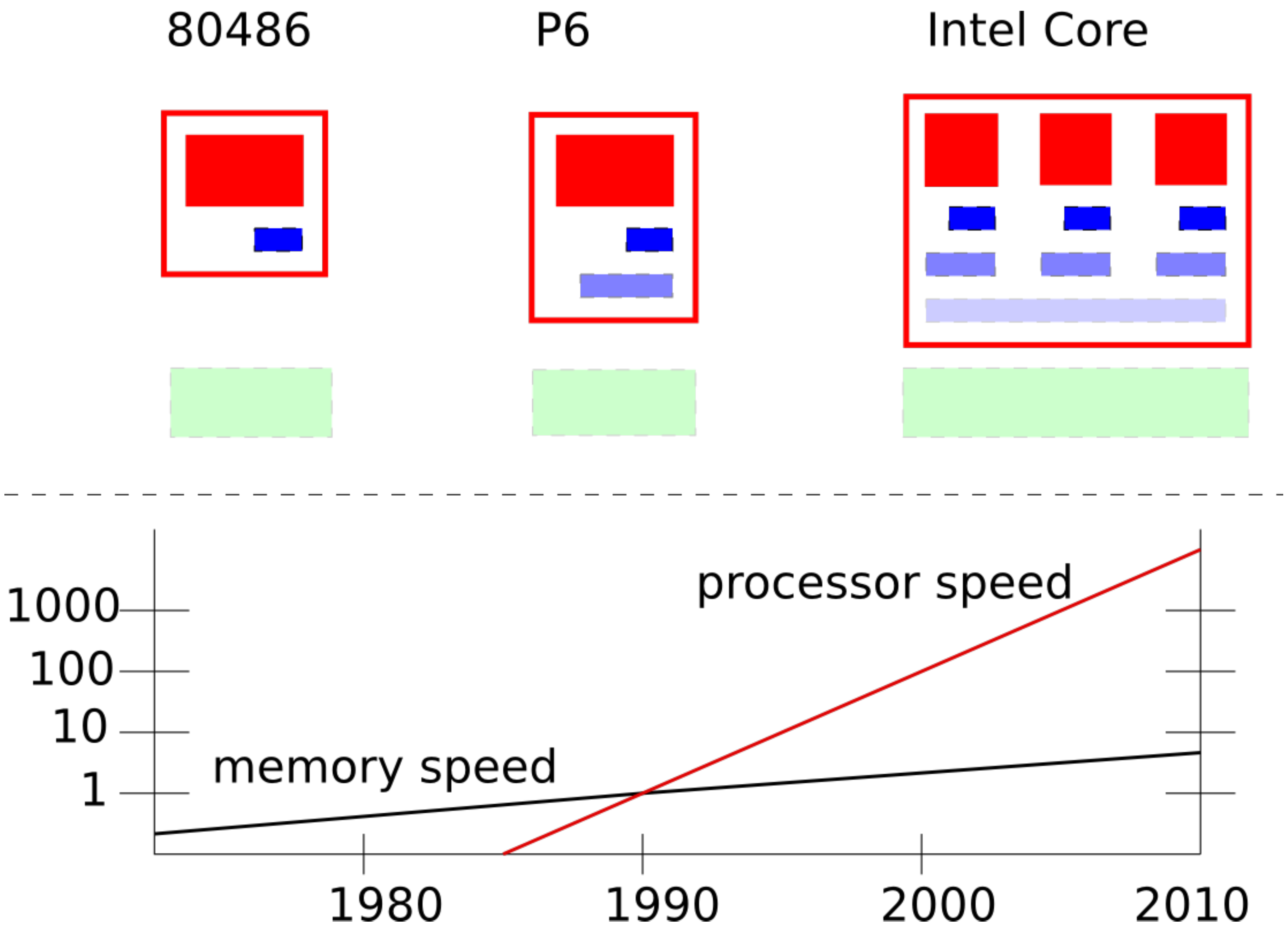
Why is C so fast anyway ?

- The compiler (gcc, icc, clang) is clever and applies many optimizations:
    - code factorization (compute only once temporary variables)
    - loop vectorization
        - exploit data parallelism
        - apply SSE,AVX instructions
        - SIMD approach
    - cache management and multithreading
- The faster the processors, the worse the memory problems

In [8]:

```
Image(filename='../code/memory_wall.png', width=200)
```

Out[8]:



Model evaluation on the CPU

Possible to do faster than C using optimizations:

- theano:
    - expression rewrite (any tensor argument)
- numexpr:
    - agressive cache management and processes balance

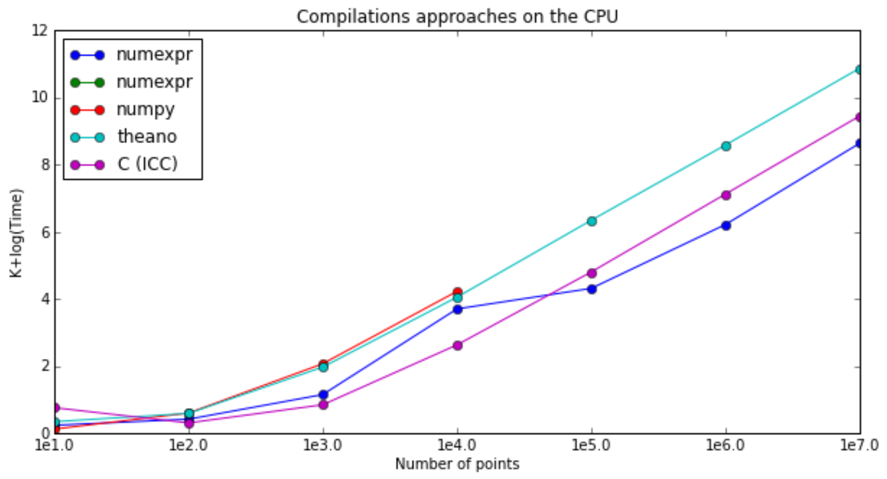
My CPU:

- Core i7 - 2670QM, 4 cores no hyperthreading

Model evaluation on the CPU

In [9]:
```
Image(filename='../code/compilations_cpu.png', width=600)
```

Out[9]:



Comparison CPU/GPU

Thanks to LLVM, JIT compilers can target the GPU

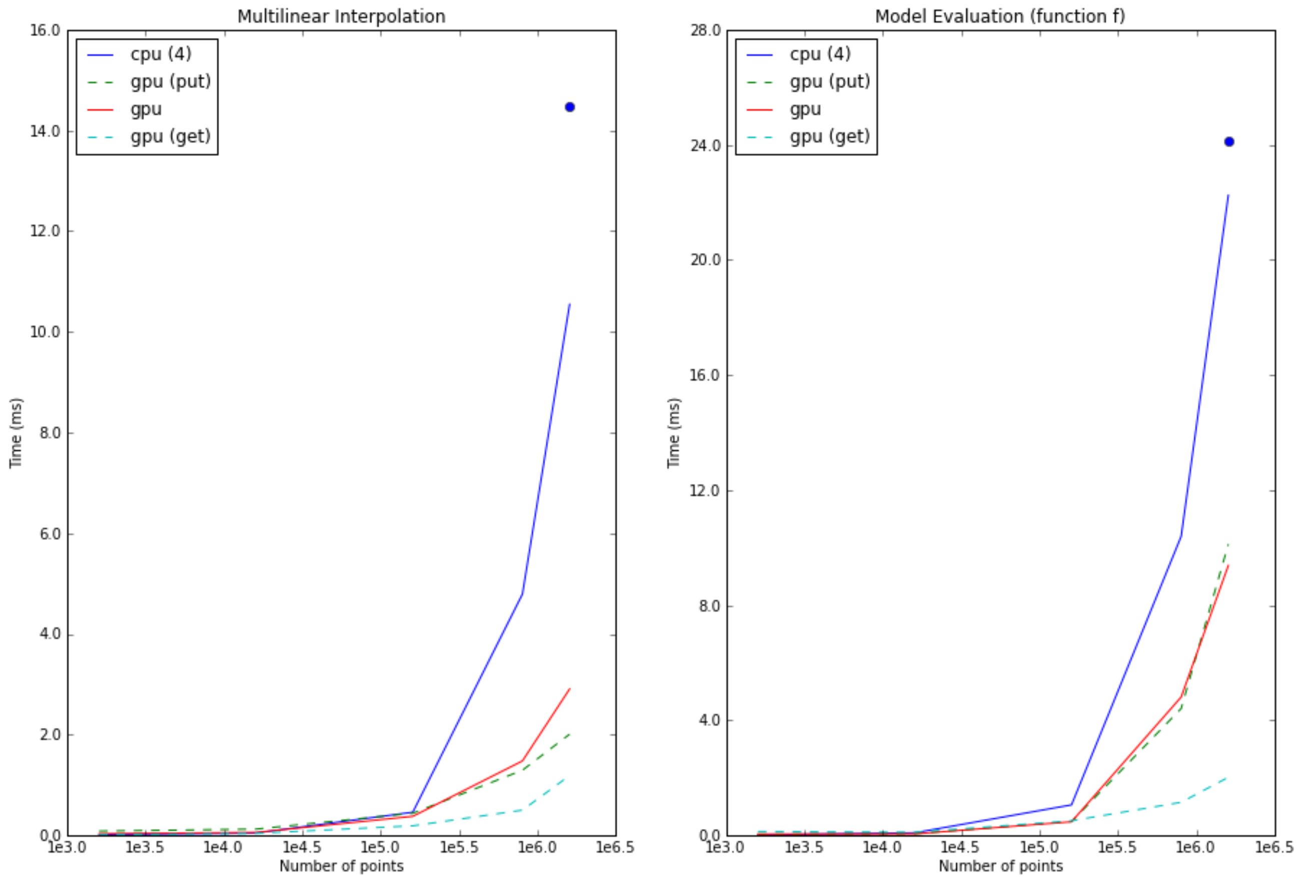
My GPU:

- Quadro 1000M, 1Gb, 96 cuda cores, no native 64 bits processing

Comparison CPU/GPU

In [10]:
```
Image(filename='../code/gpu_vs_cpu.png', width=600)
```

Out[10]:



Calibration of the RBC model

- discount factor $\beta = 0.96$
- risk aversion $\sigma = 2$
- worker's share $\alpha = 0.3$
- depreciation: $\delta = 0.1$
- output autocorrelation: $\rho = 0.9$
- output colatility $\sigma = 0.0015$

Timings for the whole solution

We compare:

- the fastest method on the CPU
- the GPU implementation (data stays on the GPU all the time)

Results

Grid size	CPU	GPU	iterations
5x32x10	0.48 s	0.63	36
5x32x100	5.77 s	3.79	37
5x32x1000	17.74 s	15.05	37

Comments:

- progression is less than proportional
  - expected for GPU
  - more suprising for CPU
- advantage for the GPU is disappointing
  - 5x32x1000 = 1.6e5
  - former slide : speed-up come from model-evaluation

Conclusion

- model standardization facilitates the design/implementation of algorithms by:
  - reusing common bricks compare to existing solution methods
- development in interpreted language/compilers, allows to play with new hardware interactively
  - conjecture: the same implementation on a modern graphic card would be very efficient
  - ready to be tested
- the natural way to think about SIMD parallelism is vectorization like in Matlab courses
- it can also benefit parallel algorithm (parallelize by chunks)
- `dolo` is free software: albop.github.com/dolo/
  - don't hesitate to try it/use it
  - it can parse a model and produce code for various output
    - matlab (compecon, or RECS)
    - julia
  - it will provide all necessary pieces to implement a solution algorithm