High-Performance Computing for Economists

An Introduction to Numba

Andrea Pasqualini

March 2020

Bocconi University

Outline

Introduction

Why do we need to speed up code?

- Economic problems nowadays get complicated very quickly
  - It is difficult to keep the no. of state variables down
  - Frictions and information issues call for complicated nonlinearities
  - More complicated nonlinearities call for denser grids for the state space
- Solving with VFI is time-consuming
- Developer time is more valuable than computing time, ...
- ...but we cannot afford to wait 3 days for solving once

Why current, high level languages are not enough?

Programming languages like Python, R and Matlab share characteristics that make them slow relative to C, C++, Fortran

- They are interpreted, as opposed to compiled
- They are weakly typed, as opposed to strongly typed
- They are single-threaded, as opposed to multi-threaded
- They operate at a higher abstraction level (i.e., they hide technicalities by making assumptions)

There are middle-ground solutions (e.g., Julia), but we can address everything with Python

Acceleration VS Parallelization

Acceleration

- Take a function, write it in a lower-level, faster language (e.g., C)
- Suitable for all functions frequently used
- Might require coding in the more complicated language

Parallelization

- Take a function, make it run on multiple cores with different data
- Suitable for all loops that are not serial in nature
- Might require coding in the more complicated language
- Requires knowledge of: thread, concurrency, parallelism (VS context switching), synchronous and asynchronous execution

We can achieve both using one Python module: **Numba**

What do we need?

For acceleration

- Not much, except for knowledge of performing language

For parallelization

- Knowledge of performing language
- Possibly a GPU

Useful concepts

- Compilers VS interpreters
  - Interpreters return the result of a program
  - Compilers return a program written in machine language (e.g., assembly)
  - Compiled programs are customized to the specific hardware and run “closer to the silicon”
- Static VS dynamic typing
  - Statically typed languages do not try to infer the type of variables: the programmer needs to declare the type, and only then can assign a (compatible) value (e.g., `int x = 1;`)
  - Weakly typed languages infer the types: the programmer just assigns values to variables, and the language heavy-lifts to infer the types from context (e.g., `a = 1.2`)
- Single- VS multi-threading
  - Single-threaded tasks run on one core, sequentially
  - Multi-threaded tasks run on multiple cores, in parallel
  - Tasks that are independent of each other can be run in parallel

Useful concepts

By choosing a compiled, statically typed language that allows multi-threading, we need to

- Compile the program (remove an abstraction level)
- Declare types (make code “less flexible”)
- Design code around tasks (make code modular)
- Potentially arrange tasks across threads

A word on design: the UNIX philosophy might help

- *Write programs that do one thing and do it well*
- *Write programs to work together*

Python decorators

Python decorators are just *syntax* that alters functions and methods

These two are equivalent

```python
@staticmethod
def f(x):
    # something
```

```python
def f(x):
    # something
f = staticmethod(f)
```

Also known as *syntactic sugar*: syntax that only improves readability, without any intrinsic added meaning

Acceleration

Acceleration

There are two main ways with Python

- Cython (`https://cython.org/`)
  - A native C extension to Python
  - Write code with Python syntax, but C semantics
  - Before execution, the Cython function is translated to C code and compiled to machine code (saved on disk)
  - Requires a C compiler on your computer
- Numba (`http://numba.pydata.org/`)
  - An implementation of the LLVM infrastructure
  - Write Python code and instruct Numba to parse it
  - During execution, LLVM compiles the function to machine code *just in time* for use (saved on memory)

Here I will show Numba. Cython can be faster than Numba if well optimized, but optimization requires a lot of manual work and quite some familiarity with the C language.

Accelerating what?

The following do the same thing: accumulate an integer from 0 to a billion

`loop.py`

```python
x = 0
i_max = 1000000000
for i in range(i_max):
    x += 1
```

`loop.c`

```c
int main () {
    int x = 0;
    int i_max = 1000000000;
    for (int i = 0; i < i_max; i++) {
        x += 1;
    }
}
```

Running in the terminal (requires Bash/Zsh)

```
$ time python3 ./loop.py
real    1m41.928s
user    1m39.734s
sys     0m0.078s
```

```
$ g++ ./loop.c  # outputs file 'a.out'
$ time ./a.out
real    0m1.708s
user    0m1.688s
sys     0m0.000s
```

Accelerating with Numba

Numba is a do-it-all module for Python that integrates with NumPy

In a nutshell

- `@jit`: do-it-all
  - Compiles the code with LLVM *just-in-time*
- `@vectorize`: write it for scalars, run it on same-size arrays
  - Numba wraps the code with efficient `for` loop for us
  - Returns a `numpy.ufunc`
- `@guvectorize`: generalization of `@vectorize`
  - Allows for passing arrays of different sizes and access elements
  - The output is passed as input
- `@stencil`: simplifies writing of stencil patterns
  - Write it with relative array indexing
  - Numba wraps it with appropriate loops
  - Numba takes care of out-of-bounds exceptions

Numba.jit

Compiles the decorated function just-in-time to machine code

Example

```
@jit
def mySum(a, b):
    return a + b
```

As simple as that? Almost...

There is a big performance difference between

`nopython` mode

```
@jit(nopython=True)
def mySum(a, b):
    return a + b
```

`object` mode

```
@jit(nopython=False)
def mySum(a, b):
    return a + b
```

Numba.jit: `nopython` VS `object` modes

`nopython` mode

- Default for `@jit`
- Avoids the Python interpreter when compiling
- Cannot deal with stuff Numba does not know about (e.g., Pandas)
- Recommended for best performance

`object` mode

- Fallback option for `@jit`
- Uses the Python interpreter when compiling
- Deals with any Python object
- Slower than `nopython` mode

Hints

- Write small, modular functions
- Optimize everything in `nopython` mode
- `@njit` is equivalent to `@jit(nopython=True)`
- Use for not-so-general functions
- General speed-ups (e.g., if you have many `for` loops)

Numba.vectorize

Numpy's universal functions, `ufuncs`, are written as if they were working with scalars, but then you can use them for arrays

Example

```
1 @vectorize(float64(float64, float64))
2 def mySum(a, b)
3     return  a + b
```

Bonuses and catches

- No need to pass in the *function signature* (but is recommended)
- Numba wraps the function in an efficient loop
- The resulting function is a Numpy `ufunc` (compare with `@jit`)
  - Broadcasting (e.g., sum between $n \times m$ and $n \times 1$ arrays)
  - Accumulation (e.g., `np.cumsum`)
  - Reduction (e.g., automatic `np.squeeze`)

Numba.guvectorize

Numpy's `ufuncs` are limited: you cannot arbitrarily mix array sizes (up to broadcasting)

Example

```
1 @guvectorize([(float64[:], float64, float64[:])], '(n), () -> (n)')
2 def scalar_multiplication(x, a, result):
3     for i in range(x.shape[0]):
4         result[i] = a * x[i]
```

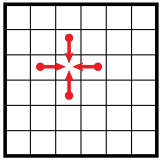
Need to indicate more structure

- Semantic output is passed as input
- Need to provide the (list of) function signature(s)
- Need to provide the *layouts* of inputs

Array	Signature	Layout
Scalar	`float64`	`'()'`
1D array	`float64[:]`	`'(n)'`
2D array	`float64[:,:]`	`'(n, m)'`
…	…	…

Numba.stencil

Stencils are computational patterns that, for every element of the input array, take elements in a neighborhood and combine them



Example

```
@stencil
def moving_average(x):  # stencil kernel
    return 0.10 * x[-2] + 0.25 * x[-1] + 0.30 x[0] + 0.25 * x[1] + 0.10 x[2]
```

Array indexing is *relative* to each element

- The stencil *kernel* is run for every element $A_{i,j}$
- `A[h, k]` within kernel actually means `A[i + h, j + k]`
- Numba takes care of looping across all elements
- Numba takes care of handling out-of-bounds exceptions

Further optimizations

- `fastmath`: accelerates purely-mathematical computations
  - Available in: `@jit`
  - Refuses to work with `NaN`'s, `Inf`'s
  - Lowers precision of floating-point calculations
- `target`: allows for specialized hardware deployment
  - Available in: `@vectorize` and `@guvectorize`
  - `target='cpu'` (default) runs code on CPU, in single-threaded mode
  - `target='parallel'` automatically uses all threads on CPU (requires function signature)
  - `target='cuda'` automatically uses an Nvidia GPU, if available (requires function signature)
- `cache`: saves compiled version on disk
  - Available in: `@jit`, `@vectorize` and `@guvectorize`
  - LLVM compilation saves compiled code to RAM by default
  - `cache=True` saves compiled code on disk, for easy reuse
  - Incompatible with `target='cuda'`

Parallelization

Parallelization

Anything that has more than one computing core supports parallelization

- CPUs nowadays are all multi-core and often support hyper-threading (i.e., one core executes two threads)
- GPUs have many more cores than CPUs, but run at lower clock frequencies
- There are many models of parallelization: here we look at *SIMD* (Same Instruction, Multiple Data)

Here I show how to parallelize on GPUs. Parallelizing on CPUs is similar in the concepts, and easy to do with Numba.

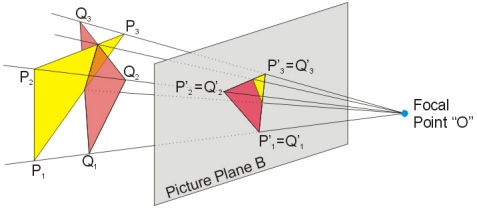
Why can GPUs help?

GPUs are typically employed in videogames

- Games graphics are typically vector graphics that need to be displayed on screen
- Screens are 2D grids of pixels and the graphics card is in charge of deciding what color each pixel should show
- Problem: “smooth” gaming performance requires the computer to refresh the screen at a rate of at least 60 frames per second
- The graphics card needs to process data from the CPU, compute all algebraic operations imposed by simulated movement (e.g., turning around in a game) and return the colors that the array of pixels should display
- Take-away: graphics cards are optimized to run *many* linear algebra operations with floating point numbers in parallel

Why can GPUs help? A graphical illustration

In graphics applications, GPUs solve a bunch of projection problems



- Desired graphic is projected onto a plane (the screen) with a set of polygons (typically, triangles)
- Movements in graphic are projected to shifts/rotations of polygon vertices

Again: GPUs are optimized to execute a ton of linear algebra operations in parallel

Why do we not use GPUs for everything?

Because they are specialized hardware, GPUs are bad for...[1]

- Everything that needs to run in serial (e.g., `for` loops that run in serial)
- Everything that requires *large* amounts of RAM
- Everything that requires parallel threads to transfer information with each other during execution
- I/O operations
- Others...

[1] https://cs.stackexchange.com/questions/121080/

How can we use GPUs?

First, a few basic concepts

- Design
- Hardware
- Terminology
- Tools

Code design

We cannot expect to just add a decorator to a Python function (as instead would be the case with `numba.jit`)

- Each computing core in the GPU can only deal with simple scalars and simple functions
- Parallelizing instructions requires us to decide how to do it
- We need to dive into a lower abstraction layer, getting more familiar with the hardware
- Depending on the application, GPU execution might add bottlenecks we might have never seen before

In short, we need to get our hands dirty

GPU hardware

A GPU is a system composed of

- Multi-core processor
  - Core: one execution unit
  - Streaming multiprocessor (SM): a set of cores
  - GPU unit: a set of SMs
- (V-)RAM module
  - Smaller RAM capacities than normal RAM
  - Much faster than normal RAM
- PCI-e interface
  - Used to communicate with the rest of the computer
  - Is subject to physical bandwidth limits
- Power supply connector
  - Delivers supply to the GPU (the GPU might be the most power-hungry component of a computer)
- Other components
  - e.g., video decoders
  - e.g., display interface (i.e., circuitry and cables to screens)

GPU hardware: example

Nvidia RTX 2080Ti[2]

- Multi-core processor
  - No. of CUDA cores: 4352
  - Core base frequency: 1350 MHz
- VRAM module
  - 11 GB
  - GDDR6: 14 Gbps
  - Bandwidth 616 GB/s
- Misc
  - Draws up to 250 W of power
  - Sustains up to 89°C
  - MSRP 1199 USD
  - Blower style air cooling

[2] https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/

Tools

The leading hardware vendor for GPU computing is Nvidia[3]

- Nvidia develops CUDA, its own C++-like programming language
- Numba has a CUDA module, so we can use Python to do GPU computing

To install all we need in Python, run in terminal

```
conda install cudatoolkit
```

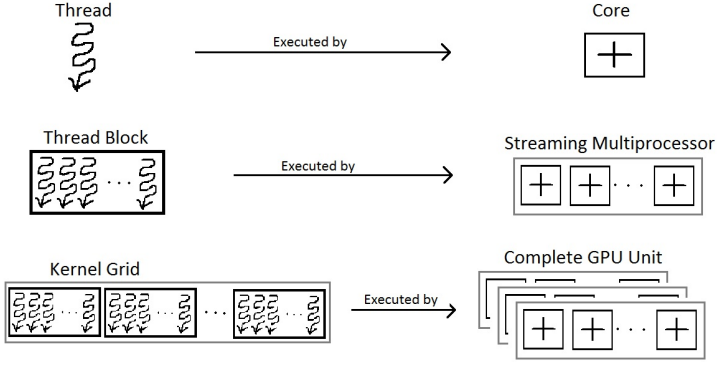
The version of `cudatoolkit` you need depends on the hardware you have: older hardware requires older version of `cudatoolkit`

[3] AMD and Intel are in the game, too, but do not really compete over GPGPU

Terminology

Host	The machine orchestrating the hardware (i.e., the CPU)
Device	The GPU unit
Kernel	The function that is deployed to the GPU for execution
Grid	A set of blocks (can organize in arrays, up-to-3D)
Block	A set of threads (can organize in arrays, up-to-3D)
Thread	The software counterpart of a processor core, i.e., a queue of instructions to execute

Relationship between hardware and software



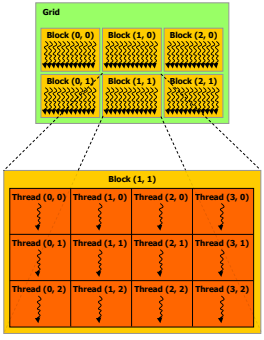
Managing threads and blocks

- The semantics of grouping threads into blocks is up to the user
- If done carefully, it allows for big performance
- Each core has access to special variables `blockIdx`, `blockDim` and `threadIdx`, with attributes `x`, `y` and `z`
  - `blockIdx` contains the coordinates of the block within the grid
  - `blockDim` contains the layout of threads inside the block
  - `threadIdx` contains the coordinates of the thread within the block
- Given SIMD paradigm, we should balance the workload uniformly across blocks and threads (i.e., do not overload one block and keep another empty)

- Example: with VFI, assign each value of the state space to one thread, maximize $V(\cdot)$ at that point within same thread

Managing threads and blocks: example

- Grid has dimension 2×3
- **Block(1, 1)** has `blockIdx.x = 1` and `blockIdx.y = 1`
- **Block(1, 1)** has `blockDim.x = 4` and `blockDim.y = 3`
- **Thread(3, 2)** has `threadIdx.x = 3` and `threadIdx.y = 2`



An example with CUDA

This CUDA code parallelizes the sum of two vectors across threads

```
void vecAddKernel (float *A , float *B , float *C , int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < n) {
        C[index] = A[index] + B[index];
    }
}
```

What are we looking at?

- `void` means that the function does not return anything
- `float` and `int` declare the type of each variable
- The semantic output `C` is passed as an input
- `blockIdx`, `blockDim` and `threadIdx` are special CUDA variables to identify the core executing the function
- Each core only operates on scalars

An example with CUDA

This CUDA code parallelizes the sum of two vectors across threads

```
void vecAddKernel (float *A , float *B , float *C , int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < n) {
        C[index] = A[index] + B[index];
    }
}
```

What is happening?

- The host transfers arrays `A`, `B` and `C` to the device memory
- Each core is assigned a thread
- Each core is aware of `blockIdx`, `blockDim` and `threadIdx`
- Cores simultaneously compute the sum $A_i + B_i$ and allocate the result C_i in the right place
- Once done, the device returns `A`, `B` and `C` to the host

Numba.cuda

Numba.cuda allows for Python code to be compiled into CUDA code

```
from numba import cuda, void, float64

@cuda.jit(void(float64[:], float64[:], float64[:]))
def sum_cuda(a, b, result):
    i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if i < a.shape[0]:
        c[i] = a[i] + b[i]

n = 640
a, b, c = np.ones((n,)), np.ones((n,)), np.zeros((n,))
threads_per_block = 32
blocks = - (-n // threads_per_block)   # ceil division
sum_cuda[(blocks, ), (threads_per_block, )](a, b, c)
```

- Calling CUDA kernel requires syntax `fun[grid_layout, block_layout](**args)`
- Managing blocks and threads is done outside kernel definition
- 640 scalar sums happen altogether: this is impossible on a CPU

GPU parallel might be slower than CPU parallel

- A big drawback for GPUs is data transfers
  - The CPU sends data to GPU
  - GPU performs computations
  - GPU returns new data to CPU
- These transfers might be a huge bottleneck
- Solution
  - Small scale problems run faster on CPU
  - Large scale problems run faster on GPU
  - Need to experiment with your hardware to find switching point

VFI benchmarks

VFI on a GPU

Can we parallelize Value Function Iteration? Yes!

- Maximizing $V(\cdot)$ at each gridpoint of the state space is independent from $V(\cdot)$ at other gridpoints

Consider the following problem

$$V(b) = \max_{c,b'} \log(c) + \beta V(b')$$
$$\text{s.t. } c + b' = y + (1+r)b$$

VFI on a GPU: example

```
@cuda.jit(void(float64[:], float64[:], float64, float64, float64))
def vmax_cuda(V, b_grid, r, y, beta):
    ix = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    VV = pwr(-10.0, 5)
    for ixp in range(b_grid.size):
        cons = (1 + r) * b_grid[ix] + y - b_grid[ixp]
        if cons <= 0:
            period_util = pwr(-10, 5)
        else:
            period_util = log(cons)
        expected = V[ixp]
        values = period_util + beta * expected
        if values > VV:
            VV = values
    V[ix] = VV
```

- The argument of `cuda.jit` is the *function signature*
- There is no explicit `max`, but `VV` will eventually be the max
- Each GPU core computes $V(b)$ at one given gridpoint b

Benchmarks: Hardware test bench

An old laptop with discrete graphics: ASUS N550JK “VivoBook Pro”

- Processor: Intel Core i7-4700HQ
  - 4 cores, 8 threads
  - Base clock frequency: 2.4 GHz
  - Boost clock frequency: up to 3.4 GHz
- RAM: 8 GB DDR3L
- GPU: Nvidia GTX 850M (640 CUDA cores @ 936 MHz each)
- OS: Ubuntu 19.10 (Linux kernel version 5.3.0)
- Price: 899 EUR (in Oct 2014)

Benchmarks: Methodology

- Solve the following ($r = 1\%$, $\beta = 0.95$, $y = 1$, tol $= 10^{-4}$)

$$V(b) = \max_{c,b'} \log(c) + \beta V(b')$$
$$\text{s.t. } c + b' = y + (1 + r)b$$

- Three functions implementing VFI
  - Unoptimized Python function on CPU
  - Jit-optimized Python function on CPU
  - CUDA/jit-optimized Python function on GPU
- Run each function N times for different values of n_b
- Compute statistics

Benchmarks: Results

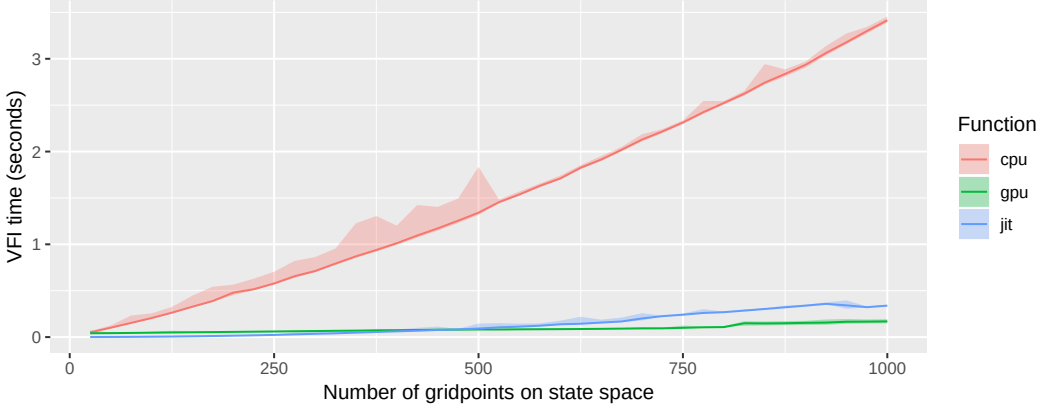


Figure 1: Averages and min/max ranges of computation times. Replications: $N = 1000$. Gridpoints on state space: $n_b \in \{25, 50, 75, \ldots, 1000\}$. Solid lines are averages. Shaded areas are min/max ranges.

Benchmarks: Results (same, in log scale)

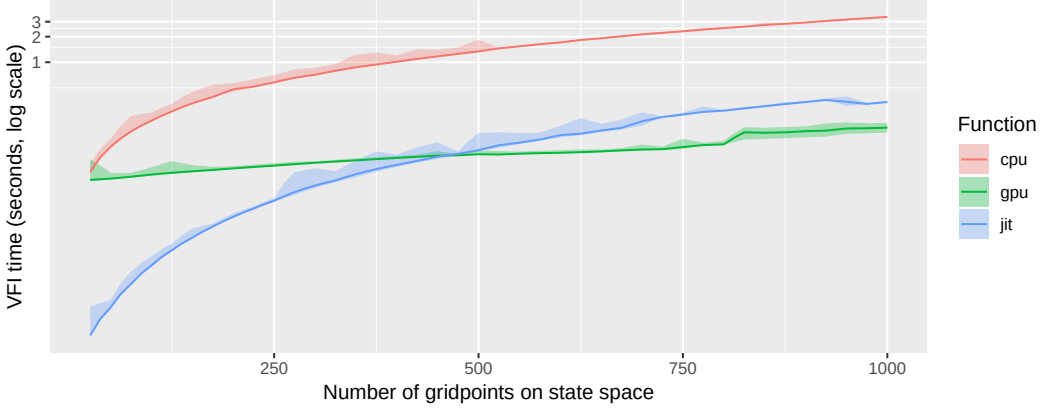
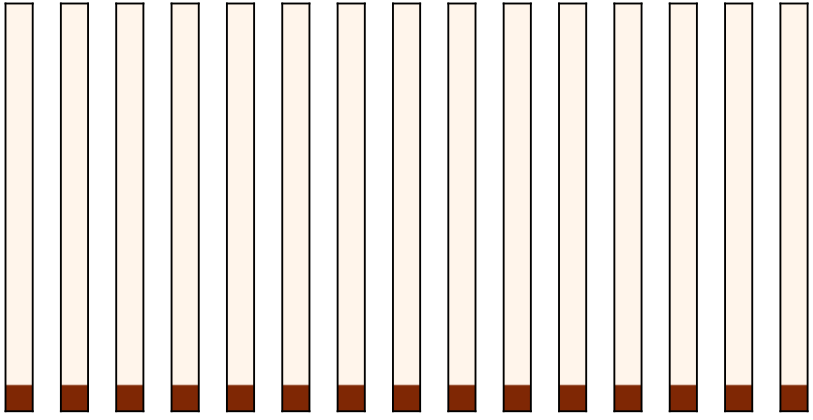


Figure 2: Averages and min/max ranges of computation times. Replications: $N = 1000$. Gridpoints on state space: $n_b \in \{25, 50, 75, \ldots, 1000\}$. Solid lines are averages. Shaded areas are min/max ranges. Vertical axis in log-scale.

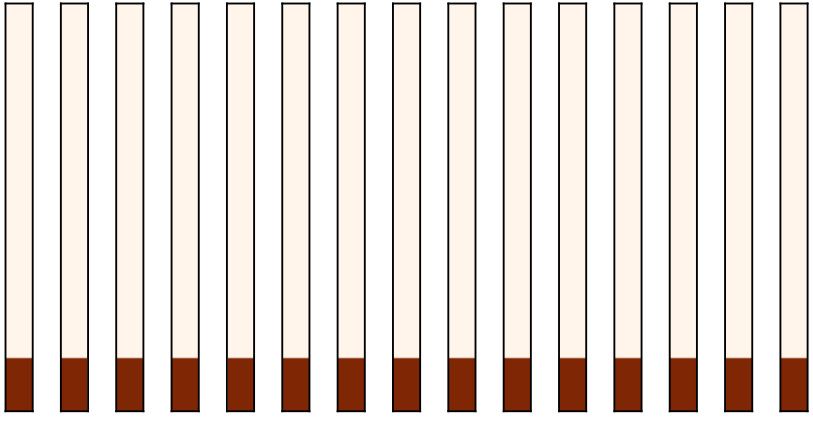
Benchmarks: Results, why GPU grows (kinda) linearly

Each bar is a block, each square in a block is a thread



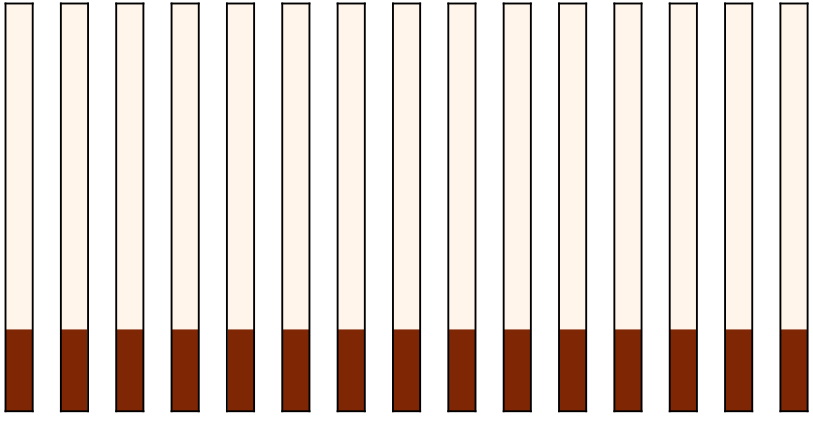
Benchmarks: Results, why GPU grows (kinda) linearly

Each bar is a block, each square in a block is a thread



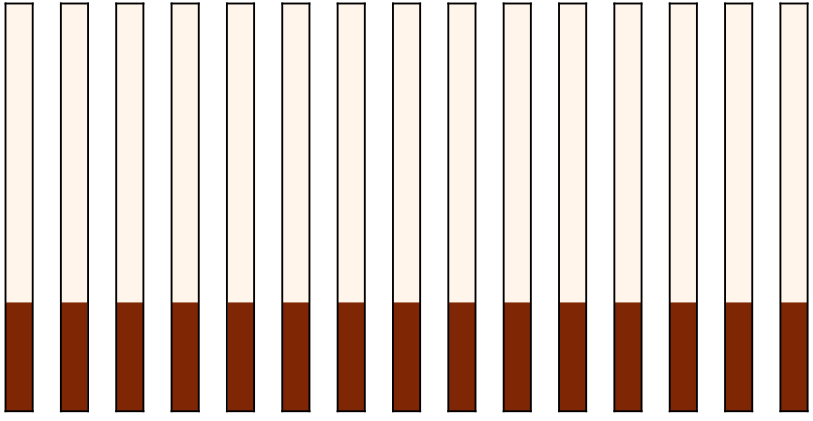
Benchmarks: Results, why GPU grows (kinda) linearly

Each bar is a block, each square in a block is a thread



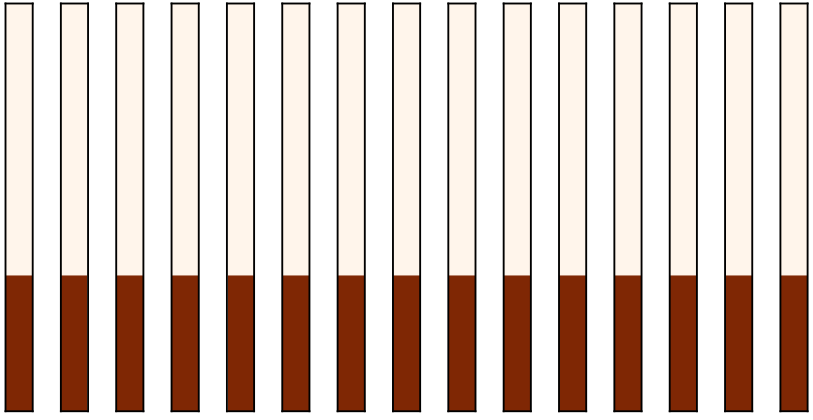
Benchmarks: Results, why GPU grows (kinda) linearly

Each bar is a block, each square in a block is a thread



Benchmarks: Results, why GPU grows (kinda) linearly

Each bar is a block, each square in a block is a thread



What to parallelize?

Suppose your code has two parallelizable steps:

- VFI to solve for the Bellman Equation
- Zero-finding routine to solve for equilibrium prices

What do we parallelize?

- It depends
- Try what is faster
- Try to match available cores to dimensions of the problem

Conclusion

Final remarks

- HPC is niche in Economics
- “Structural” people started using it
- Strongly consider accelerating code before spending money on hardware

- GPU buying considerations
  - Custom-built desktop PCs are generally cheaper than equi-potent laptops with discrete graphics cards
  - Not necessary to buy top-of-the-line, bleeding-edge hardware
  - What matters is core counts and clock speeds
  - GPUs one or two generations old are still good for us
  - GPUs last over time: think of future-proofing
  - GPU prices slowly decreasing after crypto-currency “gold rush”
  - External GPU enclosures are an option, if you don’t want to buy a whole new computer and have a laptop with a Thunderbolt 3 port

Final remarks

- Optimizing code right off the bat does not help
  - First, write your code almost as a brainstorming exercise
  - Then, **profile your code** to identify bottlenecks
  - Finally, optimize code
- A serious IDE might help (e.g., PyCharm)
- Numba's user guide is a gold mine of advice

"Premature optimization is the root of all evil."

— Donald E. Knuth