Tapping the Supercomputer Under Your Desk: Solving Dynamic Equilibrium Models with Graphics Processors

Eric M. Aldrich[1,4] Jesús Fernández-Villaverde[2]
A. Ronald Gallant[1,3] Juan F. Rubio-Ramírez[1,4]

[1]Duke University

[2]University of Pennsylvania

[3]New York University

[4]Federal Reserve Bank of Atlanta

September 30, 2010

Overview

- We show how to build algorithms that use graphics processing units (GPUs) to solve dynamic equilibrium models in economics.
- We use NVIDIA's Compute Unified Device Architecture (CUDA).
- As an example, we solve a canonical RBC model with value function iteration and document that using a GPU delivers speed improvements of around 200 times.
- GPU computing has the potential to significantly improve numerical capabilities in economics.
  - Not all economic applications are parallelizable or have the arithmetic demands to benefit from GPU computing, but the methods we discuss are broadly applicable.

Overview (Cont.)

- Likely beneficiaries of GPU computing in economics:
  - Likelihood evaluation for large sets of parameters, checking payoffs of available strategies in games, value function iteration, solving heterogeneous agents models or other rich structural models, characterization of equilibrium sets in repeated games, etc..
- GPU computing has been successfully applied in biology, engineering, statistics and weather studies, among other fields. We suggest that it will be broadly applicable to economics.

Background

- The graphics and gaming industry has fueled the development of powerful graphics processing units, each containing hundreds of individual processors.
    - These processing units were designed to handle millions of identical floating-point operations (think of rendering the same operation for millions of pixels, in parallel).
- In the mid 2000s, academic researchers involved in heavy computation discovered that they could leverage graphics processors for highly parallel algorithms.
- Accessing graphics processors was initially very difficult, but in 2007 NVIDIA released a set of development tools to allow easier accessibility to graphics hardware for general purpose computing.

Basic Algorithm

We suggest the following algorithm for a value function iteration problem:

1. Determine the number of processors, P, available on the GPU.
2. Select the number of grid points, N, and allocate them. If capital and productivity are state variables, with N_k and N_z points, respectively, then $N = N_k \times N_z$.
3. Divide the N grid points among the P processors on the GPU.
4. Make an initial guess for V^0 (a good initial guess will speed convergence).
5. Copy V^0 to the shared memory of the GPU.
6. Each processor computes V^1, given V^0, for its subset of grid points.
7. Repeat step 6 until convergence: $||V^{i+1} - V^i|| < \varepsilon$.
8. Copy V^i from GPU memory to main memory.

RBC Application: Agent

A representative household chooses a sequence of consumption, c_t, and capital, k_t, to maximize utility:

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\eta}}{1-\eta} \right] \tag{1}$$

subject to the constraint

$$c_t + i_t = w_t + r_t k_t \tag{2}$$

where β is the discount factor, η the risk aversion parameter, w_t the wage paid for a unit of labor, r_t the rental rate of capital and i_t investment.

RBC Application: Capital

The law of motion for capital is

$$k_{t+1} = (1 - \delta)k_t + i_t \tag{3}$$

where δ is the depreciation rate.

RBC Application: Firm

A representative firm has technology $y_t = z_t k_t^{\alpha}$ where productivity, z_t, evolves according to an AR(1) in logs:

$$\log z_t = \rho \log z_{t1} + \varepsilon_t, \text{ where } \varepsilon_t \sim N(0, \sigma^2). \tag{4}$$

In equilibrium $i_t = y_t - c_t$, resulting in the resource constraint of the economy:

$$k_{t+1} + c_t = z_t k_t^{\alpha} + (1-\delta) k_t. \tag{5}$$

RBC Application: Bellman

The welfare theorems hold in this economy, and hence we solve the planner's problem

$$V(k, z) = \max_{c} \left\{ \frac{c^{1-\eta}}{1-\eta} + \beta \mathbb{E}[V(k', z')|z] \right\} \tag{6}$$

subject to

$$k' = zk^{\alpha} + (1-\delta)k - c. \tag{7}$$

Dozens of other models from macroeconomics to industrial organization or game theory generate similar formulations, and our application can carry forward to those situations nearly unchanged.

RBC Application: Calibration

β	η	α	δ	ρ	σ
0.984	2	0.35	0.01	0.95	0.005

Table 1: Calibration

Implementation: Code and Hardware

- We solved Equation 6 using C++, to implement the traditional approach on a CPU. The CPU approach was not parallelized.
- We then coded the same problem in `CUDA C` in order to solve it on the GPU.
- The test machine was a DELL Precision Workstation R5400 with two 2.66 GHz quad core Intel Xeon CPUs and one NVIDIA GeForce GTX 280 GPU.
- The GeForce GTX 280 has 30 multiprocessors, each comprised of 8 processors, for a total of 240 individual cores.

Implementation: Discretization

- We discretized the productivity process with four values, using the method of Tauchen (1986).
- We forced capital to an equally spaced grid, but allowed the number of grid values to be variable.
  - We were interested in investigating the relative performance of the CPU and GPU as the capital grid became increasingly dense and to make extrapolations for asymptotically large grids.

Implementation: Maximization

- We implemented two different maximization procedures:
    1. Binary search (requires concavity of the value function)
    2. Grid search with Howard Improvement Method (only maximizing the value function every nth step, where $n = 20$).
- The CPU code exploited the monotonicity of the value function, restricting the lower bound of the maximization based on the results at previous nodes. This was not possible with the GPU, since it created dependencies that were not parallelizable.
- The Howard Improvement Method could not be used with the binary search maximization, as it did not preserve the concavity of the value function.

Results: Binary Search Solution Times

	Observed Times (seconds)					
N_k	32	64	128	256	512	1,024
GPU Memory Allocation	1.13	1.13	1.12	1.13	1.12	1.12
GPU Solution	0.32	0.36	0.4	0.44	0.57	0.81
GPU Total	1.45	1.49	1.52	1.57	1.69	1.93
CPU	0.08	0.19	0.5	1.36	3.68	10.77
Ratio (solution)	4.00	1.895	0.80	0.324	0.115	0.075
Ratio (total)	18.125	7.842	3.04	1.154	0.459	0.179
	Observed Times (seconds)					
N_k	2,048	4,096	8,192	16,384	32,768	65,536
GPU Memory Allocation	1.12	1.12	1.13	1.12	1.13	1.13
GPU Solution	1.33	2.53	5.24	10.74	22.43	47.19
GPU Total	2.45	3.65	6.37	11.86	23.56	48.32
CPU	34.27	117.32	427.50	1,615.40	6,270.37	24,588.50
Ratio (solution)	0.039	0.022	0.012	0.007	0.004	0.002
Ratio (total)	0.071	0.031	0.015	0.007	0.004	0.002

Table 2: Time to solve an RBC model using value function iteration (binary search).

Results: Binary Search Extrapolation Times

	Extrapolated Times (seconds)					
N_k	131,072	262,144	524,288	1,048,576	2,097,152	4,194,304
GPU Solution	195.767	734.498	2,843.185	11,185.46	44,369.609	176,736.311
CPU	98,362.384	392,621.758	1,568,832.79	6,272,023.978	25,081,482.86	100,312,706.6
Ratio	0.002	0.002	0.002	0.002	0.002	0.002

Table 3: Extrapolated time to solve an RBC model using value function iteration (binary search).

Results: Grid Search Solution Times

	Observed Times (seconds)					
N_k	32	64	128	256	512	1,024
GPU Memory Allocation	1.13	1.13	1.13	1.13	1.13	1.12
GPU Solution	0.16	0.20	0.25	0.24	0.53	1.50
GPU Total	1.29	1.33	1.38	1.37	1.66	2.62
CPU	0.03	0.07	0.17	0.40	1.11	3.52
Ratio (solution)	5.33	2.857	1.471	0.600	0.477	0.426
Ratio (total)	43.00	19.00	8.118	3.425	1.495	0.744
	Observed Times (seconds)					
N_k	2,048	4,096	8,192	16,384	32,768	65,536
GPU Memory Allocation	1.13	1.13	1.11	1.13	1.12	1.13
GPU Solution	4.29	15.12	57.83	222.46	875.26	3,469.78
GPU Total	5.42	16.25	58.94	223.59	876.38	3,470.91
CPU	12.52	42.85	166.43	639.89	2,527.32	10,056.00
Ratio (solution)	0.343	0.353	0.347	0.348	0.346	0.345
Ratio (total)	0.433	0.379	0.354	0.349	0.347	0.345

Table 4: Time to solve an RBC model using value function iteration (grid search).

Results: Ratios of Solution Times

	Observed Times (seconds)					
N_k	32	64	128	256	512	1,024
Ratio	10.667	5.143	2.353	1.100	0.514	0.230
	Observed Times (seconds)					
N_k	2,048	4,096	8,192	16,384	32,768	65,536
Ratio	0.106	0.059	0.031	0.017	0.009	0.005

Table 5: Ratios of solution times.

Results: Some Notes

- We interpret our results as a lower bound on the capabilities of the GPU for the problem at hand.
    - Our GPU is an off-the-shelf consumer product. There are now PCs with up to 8 NVIDIA Tesla C1060 cards, each packing 240 cores and 4 Gb of memory (our card has only 1 Gb of memory). Such a machine would speed GPU computations 8 times, or make the GPU implementation 4000 times faster.
    - NVIDIA recently released the next generation CUDA architecture (Fermi). It supports GPUs with 512 cores, allows 8 times as many double precision operations per clock cycle, increases shared memory by a factor of 4 and improves data transfer speed.
    - Our implementation of the algorithm is rudimentary. As we gain expertise, we will improve the efficiency of the GPU code.

Conclusions

- Our work does not add add any theoretically machinery to economics but is intended to introduce a computational methodology that will improve the efficiency of research.
- Computations that traditionally would have taken hours can now be performed in seconds.
  - This is significant because it allows researchers to achieve greater precision in their work or explore state spaces or models that were previous intractable.
- We believe that similar results will extend to many areas in the field of economics.