NBER WORKING PAPER SERIES

TAPPING THE SUPERCOMPUTER UNDER YOUR DESK: SOLVING DYNAMIC EQUILIBRIUM MODELS WITH GRAPHICS PROCESSORS

Eric M. Aldrich
Jesús Fernández-Villaverde
A. Ronald Gallant
Juan F. Rubio-Ramírez

Working Paper 15909
http://www.nber.org/papers/w15909

NATIONAL BUREAU OF ECONOMIC RESEARCH
1050 Massachusetts Avenue
Cambridge, MA 02138
April 2010

We thank Panayiotis Stavrinides, who first pointed out to us the potential of GPUs, Kennetz Czechowski for invaluable technical help, and the NSF for research support under several grants. The views expressed herein are those of the authors and do not necessarily reflect the views of the National Bureau of Economic Research.

NBER working papers are circulated for discussion and comment purposes. They have not been peer-reviewed or been subject to the review by the NBER Board of Directors that accompanies official NBER publications.

Tapping the Supercomputer Under Your Desk: Solving Dynamic Equilibrium Models with Graphics Processors
Eric M. Aldrich, Jesús Fernández-Villaverde, A. Ronald Gallant, and Juan F. Rubio-Ramírez
NBER Working Paper No. 15909
April 2010
JEL No. C87,E0

ABSTRACT

This paper shows how to build algorithms that use graphics processing units (GPUs) installed in most modern computers to solve dynamic equilibrium models in economics. In particular, we rely on the compute unified device architecture (CUDA) of NVIDIA GPUs. We illustrate the power of the approach by solving a simple real business cycle model with value function iteration. We document improvements in speed of around 200 times and suggest that even further gains are likely.

Eric M. Aldrich
213 Social Sciences Building,
Duke University
Durham, NC 27708
ealdrich@gmail.com

Jesús Fernández-Villaverde
University of Pennsylvania
160 McNeil Building
3718 Locust Walk
Philadelphia, PA 19104
and NBER
jesusfv@econ.upenn.edu

A. Ronald Gallant
Fuqua School of Business
Duke University
1 Towerview Drive
Durham, NC 27708
aronaldg@gmail.com

Juan F. Rubio-Ramírez
Duke University
P.O. Box 90097
Durham, NC 27708
juan.rubio-ramirez@duke.edu

1. Introduction

This paper shows how to build algorithms that use graphics processing units (GPUs) to solve dynamic equilibrium models in economics. In particular, we rely on the compute unified device architecture (CUDA) of NVIDIA. We report how this approach leads to remarkable improvements in computation time. As an example, we solve a basic real business cycle (RBC) model with value function iteration. We document how using the GPU delivers a speed improvement of around 200 times.

GPUs, a vital piece of modern computing systems,[1] are specialized processors designed to render graphics (linear algebra-like computations) for electronic games and video applications. The increasing demand for these devices, fueled by the video game industry's insatiable appetite for improved graphics processing performance, has forged a market for low-cost processing units with the number-crunching horsepower comparable to that of a small supercomputer. To illustrate this point, we report in table 1 the theoretical peak performance of two modern GPUs versus two traditional central processing units (CPUs), expressed as billions of arithmetic operations that can be computed each second (GFLOP/s), both in single and double precision.

Table 1: Theoretical peak performance of GPUs versus CPUs

	Single GFlopg/s	Double GFlopg/s
GeForce mx 280 [GPU]	933	78
Radeon HD 5870 [GPU]	2720	544
Intel Xeon E5345 (Clovertown) [CPU]	37.4	37.4
AMD Opteron 2356 (Barcelona) [CPU]	38.6	38.6

As specialized compute-intensive hardware, GPUs can devote more transistors to data processing than general purpose CPUs. This gives rise to GPU architectures with hundreds of cores (as opposed to the dual or quad core CPUs common today) with a shared memory and, therefore, well-suited to address problems that can be expressed as data-parallel computations.[2] However, since GPUs were initially designed for rendering 3D graphics and the set of instructions were specific to each particular GPU, for many years it was difficult to exploit them as general purpose computing devices.

[1] Most computers have a GPU pre-installed in the factory, either in the motherboard or in a video card.

[2] Traditional alternatives such as message passing interface (MPI) for parallel computing on many CPUs rely heavily on distributed memory. This requires the programmer to ensure that all different parts of the code running in parallel have access to the correct amount of information at the right time to avoid latency periods that diminish performance. Shared memory gets around this problem because, if the values of the relevant variables are in the shared region, they are visible to all the relevant threads.

In 2007, NVIDIA, one of the leading producers of GPUs, disrupted the supercomputing community by releasing CUDA, a set of development tools that allow programmers to utilize the tremendous computing capabilities of the GPU for general purpose computations. To date, CUDA continues to be the trend-setter and most popular implementation of this programming approach, known as graphics processing units computing (GPU computing). This innovation gives programmers access to an application programming interface (API) that allows them to easily issue and manage computations on the GPU as a data-parallel computing device without the need to understand the details of the hardware or write explicitly threaded code.

Furthermore, the CUDA development tools can be downloaded for free from the internet and installed in a few minutes on any regular computer with an NVIDIA GPU. Since CUDA programming uses `C for CUDA`, a dialect of C/C++, one of the most popular programming languages, fast code development is natural for experienced programmers. Moreover, the programming community has made available third-party wrappers in `Fortran`, `Java`, `Python`, and `Matlab` (among others), which cover all the major languages used by the scientific computing world.

The emergence of GPU computing has the potential to significantly improve numerical capabilities in economics. Although not all applications are parallelizeable or have the arithmetic demands to benefit from GPU computing, many common computations in economics fit within the constraints of the approach. For example, evaluating the likelihood function of a model for alternative parameters, checking the payoffs of available strategies in a game, and performing value function iteration are prime candidates for computation on a GPU. Over the last several decades, all of these problems have commanded the attention of researchers across different areas in economics. But even with the most updated computers, many versions of these problems, from the solution of models with heterogeneous agents to the estimation of rich structural models or the characterization of equilibrium sets of repeated games, have remained too burdensome for computation in a reasonable amount of time. GPU computing has the potential to ease many of these computational barriers.

GPU computing has already been successfully applied in biology, engineering, and weather studies, among other fields, with remarkable results. However, GPU computing has experienced a slow uptake in economics.[3] To address this void, this paper demonstrates the potential of GPUs by solving a basic RBC model. We selected this application because a common approach to solving this model is to use value function iteration, an algorithm that is particularly easy to express as a data-parallel computation. Since innumerable models from

[3] We are only aware of applications in the related field of statistics, as in Lee *et al.* (2008).

various parts of economics can be cast in the form of a dynamic programming problem, our application is representative of a larger class of situations of interest.

Our main finding is that, using value function iteration with a binary search, the GPU solves the RBC model roughly 500 times faster than the CPU for a grid of 262,144 points (65,536 points for capital and 4 points for productivity). This proves the immense promise of graphics processors for computation in economics. Parallelization, nevertheless, is less powerful in some algorithms. To illustrate these limitations, we recompute our model with a Howard improvement method and grid search. In this case, the GPU is only 3 times faster than the CPU, a noticeable improvement, but not as spectacular as before. When we let each processor use the method for which it is best suited, a difference of 200 times favors the GPU.

As we will emphasize in section 4, these numbers are a lower bound for the possible speed-ups delivered by graphics processors. First, we are using a GPU with 240 processors but there are already GPU cards with 1920 processors and larger memory available on the market (with substantially more powerful GPUs to be released in the next few months). Second, algorithm design is bound to improve with experience.

The rest of the paper is organized as follows. Section 2 describes the basic ideas of parallelization in GPUs. Section 3 presents our RBC model and the calibration. Section 4 reports our numerical results. Section 5 concludes with some final remarks and directions for future research.

2. Parallelization in GPUs

It is well known that, conceptually, it is trivial to parallelize a value function iteration. The extension to GPUs is also straightforward. A simple parallelization scheme for GPUs would be as follows:

1. Determine the number of processors available, P, in the GPU.

2. Select a number of grid points, N, and allocate them over the state space. For example, if the state variables are capital and productivity, pick N_k discrete points for capital and N_z points for productivity with $N = N_k \times N_z$.

3. Divide the N grid points among the P processors of the GPU.

4. Make an initial guess V^0. Under standard assumptions any guess will converge, but additional information such as concavity may generate a good guess that will lead to a faster solution.

5. Copy V^0 to the shared memory of the GPU.

6. Each processor computes V^1, given V^0, for its designated subset of grid points. Since the memory is shared, at the end of this step, all processors "see" V^1.

7. Repeat step 6 until convergence: $\|V^{i+1} - V^i\| < \varepsilon$.

8. Copy V^i from the GPU memory to the main memory.

While the previous algorithm is transparent, its practical coding requires some care. For example, as has been repeatedly pointed out in the parallel programming literature, we want to avoid branch instructions such as "if" statements, because they may throw the processors out of synchronization and force the code to be executed serially. In addition, to obtain a superior performance, one needs to spend a bit of time learning the details of memory management of the GPU. Since those are specific to each architecture, we avoid further discussion. Suffice it to say that, as the GPU computing technology matures, these details will become irrelevant for the average user (as they are nowadays for CPUs).

The interested reader can find the code and further implementation details at the companion web page: `http://www.ealdrich.com/Research/GPUVFI/`.

3. An Application: An RBC Model

For reasons of simplicity and generality that we outlined in the introduction, we pick as our illustrative application of the potentialities of graphics processors a basic RBC model, in which a representative household chooses a sequence of consumption c_t and capital k_t to maximize the utility function

$$\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\eta}}{1-\eta},$$

where $\mathbb{E}_0$ is the conditional expectation operation, β the discount factor, and η risk aversion, subject to a budget constraint

$$c_t + i_t = w_t + r_t k_t,$$

where w_t is the wage paid for the unit of labor that the household (inelastically) supplies to the market, r_t is the rental rate of capital, and i_t is investment. Capital is accumulated given a law of motion

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

where δ is the depreciation factor.

Finally, there is a representative firm with technology $y_t = z_t k_t^\alpha$, where productivity z_t evolves as an AR(1) in logs:

$$\log z_t = \rho \log z_{t-1} + \varepsilon_t, \text{ where } \varepsilon_t \sim \mathcal{N}(0, \sigma^2).$$

Therefore, the resource constraint of the economy is given by

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

Given that the welfare theorems hold in this economy, we concentrate on solving the social planner's problem. This problem can be equivalently stated in terms of a value function $V(\cdot, \cdot)$ and a Bellman operator

$$\begin{aligned} V(k, z) &= \max_c \frac{c^{1-\eta}}{1-\eta} + \beta \mathbb{E}\left[V(k', z') | z\right] \\ \text{s.t. } k' &= z k^\alpha + (1 - \delta) k - c \end{aligned} \tag{1}$$

that can be found with value function iteration. While, in the interest of space, we directly jump into the problem of a social planner, this is not required. The important point is that we are handling a task such as value function iteration that is inherently straightforward to paralellize. Dozens of other models, from macroeconomics to industrial organization or game theory, generate similar formulations in terms of Bellman operators. Therefore, the lessons from our application carry forward to all of these situations nearly unchanged.

Before proceeding further, we need to select values for the six parameters of our model. We pick standard numbers for a quarterly calibration. The discount factor, $\beta = 0.984$, yields a return on capital of around 6.6 percent and the capital income share, $\alpha = 0.35$, matches the observations in the data. Depreciation, $\delta = 0.01$, and risk aversion, $\eta = 2$, are conventional choices. The parameters of the AR process for productivity, $\rho = 0.95$, and $\sigma = 0.005$, match the properties of the Solow residual of the U.S. economy. Table 2 summarizes the calibration of the model.

Table 2: Calibration

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

4. Results

We coded the value function iteration that solves equation (1) in `C++` (with the GNU compiler) to implement the traditional approach on a CPU. We then coded the same problem in `C for CUDA` to solve it on a GPU with double precision. 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 composed of 8 processors, for a total of 240 cores.

We discretized the productivity process with four quadrature points following Tauchen's (1986) procedure. With respect to capital, we discretized its values for a sequence of increasingly fine grids. This helped us gauge how CUDA works with different grid sizes and to extrapolate for asymptotically large grids. We stopped at 65,536 because by that time the Euler equation errors of the approximation are sufficiently small. All the capital grids were uniform and we picked future capital points from within the grid (we also computed the case where we relax this choice by allowing interpolation outside the grid, more details below). In all of our exercises, we started with the utility of the representative household in the deterministic steady state as our V^0 and the convergence criterion was $\|V^{i+1} - V^i\| < (1-\beta)\, 1^{-8}$, where $\|\cdot\|$ is the sup norm.

For maximization, we implemented two procedures. First, as a benchmark, we employed a binary search (a method that requires concavity on the objective function). The CPU version exploited the monotonicity of the value function to place constraints on the grid of future capital over which the maximization is performed. This is not possible under the GPU version, as it creates dependencies that are not parallelizable. Our second maximization procedure was a grid search with a Howard improvement step: we maximized the value function only every $n-th$ iteration of the algorithm, where n is decided by the user (we did not rely on a Howard step for binary search since it does not preserve concavity in the steps where no maximization is performed). In our case, after some fine-tuning to optimize the performance of the algorithm, we selected $n = 20$.

Our main results appear in table 3, where we report GPU and CPU solution times (in seconds) for an increasing sequence of capital grid sizes (row N_k). We start with 16 points for capital and multiply the number by two until we have 65,536 points. The GPU method generates a timing overhead cost of approximately 1.13 seconds for memory allocation. This is the fixed cost of starting CUDA memory allocation and, hence, roughly independent of the size and quantity of objects to be allocated. For this reason, the GPU times are separated into memory allocation (second row) and solution (third row) components. The last two rows report the ratios of GPU solution time to CPU solution time and total GPU time to CPU solution time, respectively.

For coarse grids, the fixed cost of parallel programming overcomes the advantages of the GPU, but by the time there are 128 grid points of capital, the GPU starts to dominate. With 65,536 capital grid points and counting the memory allocation, the GPU is roughly 509 times faster, and, without counting it, 521 times. The key for this result is that, while the GPU computation time grows linearly in the number of grid points thanks to its massively parallel structure, the increase is exponential for the CPU, yet another manifestation of the curse of dimensionality.

Table 3: Time to solve an RBC model using value function iteration, case 1

	Observed Times (seconds)												
N_k	16	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,768	65,536
GPU Memory Allocation	1.13	1.13	1.13	1.12	1.13	1.12	1.12	1.12	1.12	1.13	1.12	1.13	1.13
GPU Solution	0.29	0.32	0.36	0.4	0.44	0.57	0.81	1.33	2.53	5.24	10.74	22.43	47.19
GPU Total	1.42	1.45	1.49	1.52	1.57	1.69	1.93	2.45	3.65	6.37	11.86	23.56	48.32
CPU	0.03	0.08	0.19	0.5	1.36	3.68	10.77	34.27	117.32	427.50	1,615.40	6,270.37	24,588.50
Ratio (solution)	9.667	4.00	1.895	0.80	0.324	0.115	0.075	0.039	0.022	0.012	0.007	0.004	0.002
Ratio (total)	47.333	18.125	7.842	3.04	1.154	0.459	0.179	0.071	0.031	0.015	0.007	0.004	0.002

In table 4, we extrapolate computation times for more dense grids. This practice, common in scientific computing, indicates how the methods would work asymptotically as we increase the number of grid points. By adjusting a simple linear regression of the square root of computation time on N_k, we guess that, for large grids, the ratio stabilizes around 0.002, or that our RBC model would take around 500 times as long to solve on the CPU as on the GPU. The adjusted R^2 values of the regressions are 0.999 (GPU) and 0.999 (CPU).

Table 4: Time to solve an RBC model using value function iteration, case 1

	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

It is important, however, to remember that some algorithms yield a lower return to parallelization. Table 5 reports the results of our same exercise, but now we employ a grid search with Howard step. This step notably reduces the length of computation time on the CPU, but not on the GPU (which actually becomes worse for large grids). Consequently, now the improvements are only 3 times. As before, we run a regression of the square of computation time on N_k to gauge the asymptotic behavior of each processor. We omit those results in the interest of space, but suffice it to say that the ratio stabilizes around 0.343.

Table 5: Time to solve an RBC model using value function iteration, case 2

	Observed Times (seconds)							Observed Times (seconds)					
N_k	16	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,768	65,536
GPU Memory Allocation	1.13	1.13	1.13	1.13	1.13	1.13	1.12	1.13	1.13	1.11	1.13	1.12	1.13
GPU Solution	0.14	0.16	0.20	0.25	0.24	0.53	1.50	4.29	15.12	57.83	222.46	875.26	3,469.78
GPU Total	1.27	1.29	1.33	1.38	1.37	1.66	2.62	5.42	16.25	58.94	223.59	876.38	3,470.91
CPU	0.02	0.03	0.07	0.17	0.40	1.11	3.52	12.52	42.85	166.43	639.89	2,527.32	10,056.00
Ratio (solution)	7.00	5.33	2.857	1.471	0.600	0.477	0.426	0.343	0.353	0.347	0.348	0.346	0.345
Ratio (total)	63.50	43.00	19.00	8.118	3.425	1.495	0.744	0.433	0.379	0.354	0.349	0.347	0.345

Finally, in table 6 we compare the ratio of times for the GPU solution with binary search and the CPU solution with grid search and Howard improvement. This gives us an idea of the speed differences when each processing unit is working with the method to which it is comparatively best suited (since the convergence criterion is very tight, the results in terms of value and policy functions are nearly identical). The ratio gives the GPU an advantage of 208 times for 65,536 capital grid points.

Table 6: Ratios of Computing Time

	Observed Times (seconds)						
N_k	16	32	64	128	256	512	1,024
Ratio	14.50	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

As we mentioned before, the results reported in tables 3-6 correspond to a solution method that constrains the values of future capital to the same grid as present capital; that is, it does not allow for interpolation. As such, the grid-based maximization procedure must evaluate the value function N_k times in order to determine a maximum. When N_k is very large, the grid-based maximization can be quite slow, especially for the GPU (relative to the CPU). An alternative solution method that we implement (results are not reported for consideration of space) fixes the grid of future capital values, independent of the grid of current capital, and evaluates the value function using piecewise linear interpolation. In our implementation, we chose the grid of future capital to have both 100 and 1,000 points, and found that the GPU was now roughly 30 times faster (in both cases) than the CPU when N_k =65,536 and roughly 40 times faster (in both cases) asymptotically. We can compare this result to the non-interpolation method (table 5), where the GPU is only about 3 times faster than the CPU.

The reason for the relatively poor performance of the GPU using the non-interpolation solution method is that the GPU loses its power as the number of serial operations on each processing unit increases. That is, for the non-interpolation method, each of the 240 GPU processors is performing roughly 65,536 serial operations for the largest grid at each step of the VFI. Compare this to the interpolation solutions, where the number of serial operations is only 100 and 1,000. Intuitively, as we increase the number of serial operations on each of the GPU processors, we are using them more and more like traditional CPUs - something for which they are not optimized. Hence, one way to improve the performance of the GPU when using a grid search for large grids is to allow for interpolation. The results may not be as striking for smaller grids, where the cost of interpolation may outweigh the benefit gained by evaluating the value function at fewer points. For the cases that we implemented (100 and 1,000 point grids for future capital), interpolation was only beneficial for the GPU when the grids for current capital had 512 and 4,096 points, respectively. The same was true for the CPU when the grids for current capital had 2,048 and 32,768 points, respectively. We note that the binary search method is not likely to enjoy the same benefits of interpolation, since the number of value function evaluations in the maximization is low and more or less fixed, independent of N_k.

We would like to emphasize that we interpret our results as a *lower* bound on the capabilities of graphics processors. Our GPU, GeForce GTX 280 with architecture GT200, is an off-the-shelf consumer product primarily geared to consumer graphics applications. In comparison:

1. Nowadays, there are PCs with up to eight NVIDIA Tesla C1060 cards. Each Tesla card packs 240 processors and a much larger memory (up to 4 Gb against the 1 Gb of the GeForce). Our reading of the CUDA documentation makes us forecast that using a eight-card machine (with 1,920 processors instead of 240) would divide computation time by eight. If our estimate turns out to be correct, the basic RBC model would take around 1,600 times as long to solve (with value function iteration) on the CPU as on eight Tesla GPUs.

2. NVIDIA has announced that it will release the next generation of CUDA architecture (codename: "Fermi") in March 2010.[4] The Graphics Fermi 100 (GF100) graphics processor displays 512 cores, delivering up to 8 times as many double precision operations per clock cycle relative to the current architecture, and allows concurrent kernel execution. The amount of shared memory per multiprocessor is 4 times as large, which can greatly minimize data transfer and speed computations (in fact, we suspect data transfer is a binding constraint for our code right now). This will produce substantial gains. More important, it demonstrates that researchers are demanding faster GPUs and that the industry will satisfy them.

3. Our version of the algorithm in the GPU is elementary, and experts in other fields have learned much about how to adapt their algorithms to achieve optimal performance from the GPUs. As economists catch up with this expertise, we foresee further improvements in speed.

[4] See `http://www.nvidia.com/object/fermi_architecture.html`

5. Concluding Remarks

This paper does not add any theoretical machinery to economics, but rather is intended to introduce readers to a computational methodology that will improve the efficiency of research. Computations that have traditionally taken hours can be completed in seconds now. This is significant because it allows researchers to calculate results to a higher level of precision or explore state spaces that were previously intractable.

There are many directions for future research. First, our intent is to rewrite our code in `OpenCL (Open Computing Language)`, a close relative of `C for CUDA` that works in a similar manner and which is also supported by NVIDIA. `OpenCL` is a framework for cross-platform, parallel programming, which includes both a language, `C99`, a modern dialect of `C`, plus APIs to define and control platforms.[5] Although, unfortunately, the current version of `OpenCL` is not object oriented, this exercise is interesting because the new programming language is quickly expanding within the industry. A second avenue for research is to test how GPUs work for other types of algorithms commonly used in economics, such as projection or perturbation methods. We hope additional findings regarding these questions will be forthcoming soon.

References

[1] Lee, A., C. Yau, M.B. Giles, A. Doucet, and C.C. Holmes (2008). "On the Utility of Graphic Cards to Perform Massively Parallel Simulation with Advanced Monte Carlo Methods." *Mimeo*, Oxford University.

[2] Tauchen, G. (1986), "Finite State Markov-chain Approximations to Univariate and Vector Autoregressions." *Economics Letters* 20, 177-181.

[5] See `http://www.khronos.org/opencl/`.