Econometrics on GPUs

Michael Creel[a,1], Sonik Mandal[b], Mohammad Zubair[b]

[a] *Universitat Autònoma de Barcelona, Barcelona Graduate School of Economics and MOVE*
[b] *Old Dominion University*

Abstract

A graphical processing unit (GPU) is a hardware device normally used to manipulate computer memory for the display of images. GPU computing is the practice of using a GPU device for scientific or general purpose computations that are not necessarily related to the display of images. Many problems in econometrics have a structure that allows for successful use of GPU computing. We explore two examples. The first is simple: repeated evaluation of a likelihood function at different parameter values. The second is a more complicated estimator that involves simulation and nonparametric fitting. We find speedups from 1.5 up to 55.4 times, compared to computations done on a single CPU core. These speedups can be obtained with very little expense, energy consumption, and time dedicated to system maintenance, compared to equivalent performance solutions using CPUs. Code for the examples is provided.

Keywords: parallel computing; graphical processing unit; GPU; econometrics; simulation-based methods; Bayesian estimation.

JEL codes: C13, C14, C15, C33.

1. Introduction

A graphical processing unit (GPU) is a hardware device normally used to manipulate computer memory for the display of images. GPUs have evolved to have a great capacity for floating point computations of the sort that are needed to display images, including applications of shading and textures, geometrical computations related to the rotation of objects, interpolation, etc. GPUs achieve their exceptional performance for these operations by performing computations in parallel, using a large number of cores. While a typical desktop or laptop computer may have a 2 or 4 core CPU, the GPU that it contains may have dozens, hundreds, or even thousands of cores.

Email addresses: `michael.creel@uab.es` (Michael Creel), `smand009@odu.edu` (Sonik Mandal), `zubair@cs.odu.edu` (Mohammad Zubair)

[1]Support from grants MICINN-ECO2009-11857 and SGR2009-578 is gratefully acknowledged.

GPU computing is the practice of using a GPU device for scientific or general purpose computations that are not necessarily related to the display of images. The idea is to take advantage of the many cores of the GPU to accelerate computations by offloading part of the work of the CPU to the GPU. Not all problems are good candidates for GPU computing. Certain computations exhibit data-level parallelism, which means that the same instructions are applied to many different pieces of data. An example is the computation of a log-likelihood function, where the log-likelihood is computed for each sample observation, and then summed up. In general, GPUs are well-suited to data parallel problems. However, in a data parallel environment, it is possible for parallel execution units to perform different operations on their data, based upon logical branching (i.e., "if" statements). GPU hardware realizes full efficiency when all threads in a group take the same execution path. Logical branches resulting from "if" statements can result in "thread divergence", where threads within a group take different execution paths. Problems which cannot avoid excessive thread divergence are likely not good candidates for porting to the GPU. Another consideration is that GPU computing requires transferring data from the host computer to the GPU device, and results from the GPU device back to the host computer. Such memory transfers are relatively slow. A problem that is a good candidate for GPU computing should exhibit data parallelism, with high computational intensity relative to required memory transfers, and should require little logical branching. Within econometrics, many problems fit these requirements reasonably well. This paper explores several examples of econometric computations that can beneficially be ported to GPU computing.

To give a bit of background, interest in scientific uses of GPU computing began in the early 2000s. In 2006 and 2007, Nvidia introduced the CUDA parallel computing architecture and software development kit, which allowed programming for GPUs using an extension of the C++ language. This programming model made GPU programming much more accessible than it had been previously, and it has since been used in many applications in many areas, as perusal of the web page `http://www.nvidia.com/object/cuda_showcase_html.html` reveals. Speedups reported on that page range from 1 to 3 orders of magnitude, compared to computations done using a single CPU thread. CUDA and the associated programming environment generates code that runs only on hardware devices from Nvidia. The OpenCL[2] programming framework is a standard governed by the Khronos non-profit consortium, and it provides a GPU computing framework similar to CUDA. The OpenCL language and application programming interface allows for GPU programming and mixed CPU/GPU programming using hardware devices from a number of manufacturers, including Nvidia, Intel and AMD. Though OpenCL and other alternatives to CUDA are available, the CUDA framework is at present the most widely used environment for GPU programming. It is the environment for which most examples available, and there exists an extensive set of code libraries that can facilitate GPU program-

[2] `http://www.khronos.org/opencl/`

ming using CUDA. Libraries exist for basic linear algebra subroutines (cuBLAS) and fast Fourier transform (cuFFT), for random number generation (cuRAND), and to enable functionality of the C++ standard template library on the GPU (Thrust). As well, third parties have developed extensions and wrappers for many widely used programming languages, including Python, Matlab, Mathematica, and others. Because of the completeness of the environment and its ease of use at the present time, we focus on uses of CUDA and related technology in this paper. However, it is worth emphasizing that GPU computing is evolving rapidly and is continually becoming more accessible. We expect that CUDA and its alternatives will continue to become easier to use.

Another factor which makes learning about GPU computing attractive is that the computational capacity of GPU hardware is constantly increasing at a rapid rate, as is its energy efficiency. From the page `en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units`, we can trace the performance of GPUs offered for desktop computers. Taking as an example mid-level cards, which offer good performance at a moderate price, the Nvidia GTX460 GPU was introduced in July 2010, and offered 907 Gflop/s and 6.05 Gflop/s per watt of power consumption, for single precision computations. The GTX560 Ti was introduced in January 2011 and offered 1263 Gflop/s and 7.43 Gflop/s per watt. At the time of this writing (March, 2012), the GTX460 is available for roughly $150, while the GTX560Ti costs roughly $230 (prices from `www.amazon.com`). These theoretical peak numbers for single precision computations may not translate directly to the performance one sees in real-world applications, but they do illustrate the evolution of GPU computing power and energy efficiency over time.

For a more realistic comparison of the power of GPU computing in relation to CPU-only computing, Phillips (2010) reports results for the widely-used LINPACK benchmark, using double precision. A CPU-based system achieves 11 Gflop/s per $1000 cost, and 0.15 Gflop/s per watt of power consumption. A mixed CPU-GPU system achieves 60 Gflop/s per $1000 cost, and 0.66 Gflop/s per watt. It is clear that GPU computing adds a great deal of performance relative to cost or energy consumption.

In spite of the potential for cheap access to this source of computing power, there has been, up to the present time, remarkably little use of GPU computing in research work in economics and econometrics, though other fields such as statistics (e.g., Suchard et al. 2010) and areas in the biological sciences (e.g., Liepe et al. 2010) have seen more work done. Mathur and Morozov (2012) use CUDA to solve an optimal control problem that involves experimentation and learning, using value function iteration, and report speedups of 15-26 times. Aldrich et al. (2011) show how GPU computing can be used to accelerate the solution of a dynamic equilibrium model, also using value function iteration and CUDA. They report speedups of around 200X. Durham and Geweke (2011) present an algorithm for Bayesian estimation of models, using sequential Monte Carlo (particle filtering), implemented with CUDA. They do not compare their CUDA software to a CPU version. Li (2011) uses CUDA to implement kernel density estimation on a GTX 470 device, which has 448 processor codes, and reports speedups of more than 400X compared to single threaded Matlab code.

The very limited use of GPU computing in economics and econometrics up to the present time can probably be explained by the more demanding computer programming skills that are needed to code GPU applications. One factor that can help to overcome this barrier is the availability of working and clearly documented code examples in areas relevant to economists and econometricians. This paper provides some examples of GPU computing applied to econometric estimation, accompanied by source code that is documented and explained. The intention is not to provide an exhaustive survey of potential uses, but rather to provide some working examples that illustrate the potential of GPU computing for econometric estimation. The source code that accompanies the paper can serve as a model for development of code for other estimation problems.[3]

We present two examples. The first is the computation of a likelihood function. This is a simple example that illustrates the basic ideas, it has application in both classical and Bayesian econometrics, and it shows that a good speedup can be obtained. The second example uses the indirect likelihood estimators of Creel and Kristensen (2011). These estimators are simulation-based, and simulation is a data parallel task that is an ideal candidate for porting to the GPU. We find that use of GPU computing can deliver results up to 55 times more quickly than is possible using a single CPU core. We expect that similar speedups would apply to many other problems in econometric estimation.

2. Examples

2.1. Likelihood function

For a sample $Z_n = \{(y_t, x_t)\}_n$ of n independent observations of a dependent variable (y_t) and vector of explanatory variables (x_t), let $f(y_t|x_t;\theta)$ be the density of y_t conditional on x_t, with parameter vector θ. The maximum likelihood estimator of the parameter θ is $\hat{\theta} = \arg\max \sum_{t=1}^{n} \ln f(y_t|x_t,\theta)$. In the context of seeking to parallelize computations, Swann (2002) and Creel (2005) point out that the sum can be broken into sums over blocks of observations, using up to n blocks. For parallelization using MPI, one would normally use as many blocks as available CPU cores. With a many-core GPU device, a larger number of blocks may be used, potentially giving a greater speedup. The simple presentation used here can easily be adapted to dependent observations, and the basic idea of computing the likelihood using blocks of observations is possible as long as the data are Markovian.

To give a concrete example, suppose that y_t is conditionally distributed Poisson:

$$\begin{aligned} f(y_t|x_t) &\sim \frac{\exp(-\lambda_t)\lambda_t^{y_t}}{y_t!} \\ \lambda_t &= \exp(x_t'\theta) \end{aligned}$$

[3]The code is archived at pareto.uab.es/mcreel/GPU_paper_code_release_v1.zip

where x_t and θ are k vectors. In this case, the conditional log-likelihood function is $\ln f(y_t|x_t,\theta) = -\exp(x_t'\theta) + y_t x_t'\theta - \log(y_t!)$. Computation of the maximum likelihood estimator requires iterative maximization, involving a number of evaluations of the objective function. Likewise, Bayesian estimation using Markov chain Monte Carlo requires many evaluations of the likelihood function (among other computations) as the chain advances, where the likelihood function may be computed as

$$L(\theta; Z_n) = \exp\left(\sum_{t=1}^{n} \ln f(y_t|x_t,\theta)\right).$$

Focusing on this last case, we will look at the time to perform 2000 evaluations of the Poisson likelihood function, when the dimension of x_t and θ is 3. Each of the 2000 evaluations is done at a different value of the parameter vector, as would be the case if MCMC were being done. Furthermore, the value of the parameter is determined using CPU computations, and a memory transfer communicating the new parameter values from the CPU to the GPU is done at each step. On the other hand, the data, Z_n, which uses much more memory, must only be transfered to the GPU once. This coding solution allows GPU computation of the likelihood to be dropped in as a replacement for CPU-based computation, into relatively complex CPU-based software for estimation by maximum likelihood or MCMC. The computation of the likelihood function is the inner-loop computational bottleneck. When dealing with a complex software chain, optimizing the inner loop is often where most gains can be made. It may not be desirable to port an entire complex software chain to GPU computing if substantial gains can be made by optimizing only the likelihood function computations. This example explores this idea.

2.2. Indirect likelihood inference

Indirect likelihood inference (Creel and Kristensen, 2011) is method of econometric estimation that relies on simulations from the model and on nonparametric density or regression function computations. A very similar, and in some aspects, identical, class of estimators is known in the literature of the biological sciences as Approximate Bayesian Computation (ABC) or likelihood-free Bayesian inference (see, e.g., Tavaré *et al.,* 1997; Beaumont, Zhang and Balding, 2002; Marjoram *et al.,* 2003; Sisson, Fan and Tanaka, 2007). The paper of Creel and Kristensen makes clear the relationship with maximum likelihood estimation, and establishes a theoretical base for the estimators. Here, we follow the notation of that paper. The combination of simulation and nonparametrics means that the estimators can be computationally demanding. Because simulations are independent of one another, the needed computations can easily be parallelized. Likewise, nonparametric methods such as kernels and nearest neighbors require computing distances between large sets of points, and this is also easily parallelized (Racine, 2002; Creel, 2005; Garcia, 2008; Garcia, Debreuve and Barlaud, 2008).

First, let us briefly describe the estimators. We wish to learn about a parameter $\theta \in \Theta \subset \mathbb{R}^k$ describing a model. Given a sample $Y_n = (y_1, ..., y_n)$

from the model, we make inference on θ through a q-dimensional statistic, $Z_n = Z_n(Y_n) \in \mathbb{R}^q$. Let $f_n(Z_n|\theta)$ be the likelihood of the statistic for a given value of the parameter. Ignore for a moment the fact that the likelihood is normally not known on closed form. The maximum indirect likelihood (MIL) estimator maximizes the indirect likelihood defined through Z_n:

$$\hat{\theta}_{MIL} = \arg\sup_{\theta\in\Theta} \log f_n(Z_n|\theta). \tag{2.1}$$

This is very much like a maximum likelihood estimator, except that the sample is filtered through a statistic. This has the advantage of reducing the dimension of random quantities from $O(n)$ to the finite value q, which facilitates the use of nonparametric methods. The disadvantage is a potential loss of efficiency if Z_n is not a sufficient statistic.

A Bayesian version of the MIL estimator may be of interest, following considerations in Chernozhukov and Hong (2003), as it obviates the need for numerical optimization. One possibility is to use the posterior mean of θ given Z_n defined as

$$\hat{\theta}_{BIL} = E(\theta|Z_n) = \int_\Theta \theta f_n\left(\theta|Z_n\right) d\theta, \tag{2.2}$$

where $f_n\left(\theta|Z_n\right)$ is the posterior distribution given by

$$f_n\left(\theta|Z_n\right) = \frac{f_n\left(Z_n,\theta\right)}{f_n\left(Z_n\right)} = \frac{f_n(Z_n|\theta)\pi\left(\theta\right)}{\int_\Theta f_n(Z_n|\theta)\pi\left(\theta\right) d\theta} \tag{2.3}$$

for some pseudo-prior density $\pi(\theta)$ on the parameter space Θ. We refer to this particular estimator as the Bayesian indirect likelihood (BIL) estimator. For most choices of Z_n, the density $f_n(Z_n|\theta)$ is of unknown form, so the MIL and BIL estimators are infeasible. Feasible versions can be computed using simulation and nonparametric estimation. Here we discuss only a feasible version of the BIL estimator, for the feasible version of the MIL estimator, and additional discussion, see Creel and Kristensen, 2011.

2.2.1. SBIL

The BIL defined above (equation 2.2) is a posterior mean. The SBIL estimator proposed in Creel and Kristensen (2011) (essentially the same idea was proposed in the ABC literature by Beaumont, Zhang and Balding, 2002) directly computes the posterior mean using simulation and nonparametric regression, as follows. Make i.i.d. draws θ^s, $s = 1, ..., S$, from the pseudo-prior density $\pi(\theta)$, for each draw generate a sample $Y_n(\theta^s)$ from the model at this parameter value, and then compute the corresponding statistic $Z_n^s = Z_n(Y_n(\theta^s))$, $s = 1, ..., S$. Now let $\mathcal{Z}_S = \{Z_n^s,\, s = 1, 2, ..., S\}$ be the set of the simulated statistics. We can obtain a simulated version of the BIL (SBIL) through nonparametric regression techniques. One such is a simple K nearest neighbor regression estimator (see Li and Racine, 2007, Ch. 14),

$$\hat{\theta}_{SBIL} = \frac{1}{K}\sum_{s=1}^{S} \theta^s \mathbf{1}\left(\|Z_n^s - Z_n\| \le d_K(Z_n, \mathcal{Z}_S)\right), \tag{2.4}$$

where $\mathbf{1}(\cdot)$ is the indicator function that take the value 1 if the argument is true, and the value 0 otherwise, and $d_K(Z_n, \mathcal{Z}_S)$ is the Euclidean distance between Z_n and the K^{th} closest element of $\mathcal{Z}_S$ to Z_n. Simply put, this estimator is the average of the K values of θ^s that lead to the K closest neighbors to Z_n. There are more sophisticated possibilities using weighting schemes, but this simple version presents the main ideas clearly. This important point is that this estimator is consistent for the posterior mean $E(\theta|Z_n)$ as S increases, as long as K is chosen to be an appropriately slowly growing function of S. Because S is the number of simulations and can be chosen, we can make the nonparametric approximation to the true posterior mean as accurate as is needed by using a sufficient number of simulations.

2.2.2. Two example models for IL estimation

MA model. A first model, chosen for it's simplicity, is the first order moving average (MA(1)) model

$$\begin{aligned} y_t &= \epsilon_t + \psi\epsilon_{t-1} \\ \epsilon_t &\sim i.i.d.\, N(0, \sigma^2) \end{aligned}$$

We use a sample sizes of n=200 observations. The parameter ψ is one of the values $\{-0.95, -0.9, -0.5, 0, 0.5, 0.9, 0.95\}$, so the model is always invertible. The parameter σ is always equal to 1. The parameter vector is $\theta = (\psi, \sigma)$. We set the parameter space to $\Theta = (-1, 1) \times (0, 2)$, which imposes invertibility, which is needed for the parameter to be identified. The auxiliary statistic Z_n is the vector of estimated parameters $(\rho_0, \rho_1, ..., \rho_P, \sigma_v^2)$ of a P-order autoregressive (AR(P)) model $y_t = \rho_0 + \sum_{p=1}^{P} \rho_p y_{t-p} + v_t$, fit to the data using ordinary least squares. For simplicity, we hold the order of the AR(P) model constant at $P = 10$ across the Monte Carlo replications. Thus, the dimension of Z_n is 12, while the dimension of θ is 2, so we have considerable overidentification.

Structural auction model. One would not normally estimate an MA(1) model using a simulation-based estimator such as those discussed in this paper. However, rich structural models with latent variables and nonlinearities often require the use of simulation-based estimators. An example is the structural auction model presented by Li (2010), who studies the performance of the indirect inference estimator using Monte Carlo. Creel and Kristensen (2011) replicate the Monte Carlo study, using the SBIL estimator. Here, we port the same example to the GPU.

The model is a Dutch auction, where only the winning bid is observed. The number of bidders is fixed at $N = 6$, and the sample size is $n = 100$, meaning that the outcomes of 100 auctions are observed. At each auction $i = 1, 2, ..., 100$, the quality, x_i, of the item being auctioned is the square of a uniform $(0, 2)$ random variable, to introduce heterogeneity in the values of the objects across the auctions. The 6 bidders draw their independent private values from a common exponential distribution with density

$$f(v|x_i) = \frac{1}{\exp(\theta_0 + \theta_1 x_i)} \exp\left(-\frac{v}{\exp(\theta_0 + \theta_1 x_i)}\right)$$

so that $\exp(\theta_0 + \theta_1 x_i)$ is the mean valuation of the item, over the bidders. The equilibrium strategy for the winning bid is

$$b_i^* = v_i^* - \frac{1}{F^{N-1}(v_i^*|x_i)} \int_0^{v_i^*} F^{N-1}(u|x_i)du$$

where v_i^* is the highest private valuation, and $F(\cdot|x_i)$ is the exponential distribution function. For a given value of N (6 in this case), symbolic computation software can be used to obtain a compact analytic solution for the winning bid, so the model is easily simulated. The observed data are the 100 values of $\{x_i, b_i^*\}$, and we seek to estimate θ_0 and θ_1. The true values are set to $\theta_0 = 1$ and $\theta_1 = 0.5$. To define the auxiliary statistic, we fit the auxiliary model $\log b_i^* = \alpha + \beta x_i + \sigma\epsilon_i$ using ordinary least squares. The auxiliary statistic is $Z_n = (\widehat{\alpha}, \widehat{\beta}, \log\widehat{\sigma})$. The parameter space, for reasons discussed in Creel and Kristensen (2011), is set to $(\theta_0, \theta_1) \in\Theta = (-0.05, 2.45) \times (0.00, 1.96)$. The short explanation is that values outside this region never generate points close to a Z_n that is generated at the true value.

3. The Code and benchmarking

3.1. The code

We provide Matlab/Octave and CUDA code to compute the Poisson likelihood function and to estimate the MA and auction models by SBIL. This code is documented and commented, so we do not describe it in detail here, except to note some points of interest. The CUDA code for the Poisson likelihood is very simple, and serves as a tractable introduction to econometrically relevant computations using CUDA. It makes some use of the Thrust library. For the SBIL examples, both the MA and auction example use an ordinary least squares (OLS) fit as the auxiliary statistic, and at the CUDA kernel level there are no standard libraries available for OLS. We wrote a CUDA code library for OLS using Hall (1970) as a guide. This part of our code may be of interest beyond the examples we work with. Finally, the SBIL estimator uses nearest neighbors nonparametric regression to approximate the conditional mean (see equation 2.4). To perform these computations on the GPU, we use an existing implementation available at `http://www.i3s.unice.fr/~creative/KNN/` (Garcia, 2008; Garcia, Debreuve and Barlaud, 2008). This code uses the brute force method of computing nearest neighbors, which involves explicitly computing the distances between all target and query points. We have modified this code to use multiple GPU devices, if available.

3.2. Benchmarking

3.2.1. Poisson likelihood function

We computed 2000 iterations of the Poisson likelihood function, for sample sizes $n \in \{2^8, 2^{10}, ..., 2^{16}\}$ (for reference, $2^8 = 256$, and $2^{16} = 65536$). In all cases, there are 3 regressors. The CPU-based computations were done using Matlab

R2010a, running on a Macbook Air laptop with an Intel Core i5-2557M CPU, running at 1.7 GHz. The computations using the GPU were done on a single GTX560Ti card, which has 384 CUDA cores and 1GB of memory. The retail price of this card is presently around \$230 (price from `www.amazon.com`). The sample data is transfered to the GPU only once, and then the likelihood function is evaluated at each of 2000 different parameter values, where the parameter values are determined sequentially using CPU-based computing. Thus, there are 2000 CPU->GPU memory transfers of the value of the parameter vector θ during the course of the computations, exactly as would occur if the GPU-based computation of the likelihood were embedded in a CPU-based Markov chain Monte Carlo (MCMC) or iterative maximization program. The timing results are found in Table 1. In this and other tables, $\log_2 S$ means the base-2 logarithm of S, so that $S = 2^{\log_2 S}$. Thus, when $\log_2 S$ increases by 1, S doubles. The timings for the CUDA runs include all memory transfers, that of the data, and each of the 2000 parameter vector values. For the smallest sample size, we see that the computations using the GPU device are about 1.5 times faster than the computations done using Matlab on a laptop computer. For the largest sample size, the computations on the GPU are almost 18 times faster. The greater speedup for the larger sample size is certainly influenced by the greater computational intensity relative to the time needed for the memory transfers, which is constant with respect to the sample size. These results show that even simple computations with a small sample size can be accelerated using CUDA, and that when the sample size becomes larger, the acceleration becomes considerable. For a more computationally intensive likelihood function, we expect that the speedup would be greater still.

3.2.2. SBIL estimator

For the SBIL estimator, the basic factor that determines execution time is the time needed to compute S simulations of the auxiliary statistic and to find the K nearest neighbors, among the S simulated values, to the observed value of the auxiliary statistic, Z_n. In this section we report timings for the two models, along with timings for the same computations done on a CPU. The CPU-based computations were done using a single core of an Intel Core 2 Duo E8400 CPU, which has a clock speed of 3.0 GHz. For the CPU computations, the simulation loop was programmed using Matlab R2010a. The nearest neighbors part of the computations on the CPU were using the well-known ANN library (`http://www.cs.umd.edu/~mount/ANN/`). It should be noted that the simulation part of the problem is trivial to parallelize, and that the simulation time greatly dominates the time to find the nearest neighbors, so roughly speaking, one could reduce the CPU time by a factor of X by running the code on X homogeneous cores, either on a single machine or on a cluster.

The time to find the K nearest neighbors is virtually constant with respect to K, though it is sensitive to S. This is because to find the nearest neighbor to Z_n in the set of S points, $\mathcal{Z}_S$, or the K nearest neighbors, all of the S distances must be computed in both cases. For this reason, we set K to a single value, 50. Also, it is clear that timings will scale linearly in S, as we simply need to generate proportionally more auxiliary statistics and compute proportionally more distances. For this reason, we present results for several moderate values of S, but we do not use very large values of S, because the CPU timings would become large, without contributing any useful additional information. Regarding the two models, for the MA model, the auxiliary statistic has dimension 12, while for the auction model, the dimension is 3. The sample size for the MA model is 200, while for the auction model it is 100. Overall, the MA model involves manipulating considerably more data, but the computations are very simple. The auction model involves less data, but more complex computations.

Table 2 presents timings for the MA model, for $S = 2^{17}$ up to $S = 2^{20}$ ($2^{17} = 131,072$ and $2^{20} = 1,048,576$). Timings are given as total wall clock time, measured using the Linux "time" command on an otherwise unloaded system. We can observe that the CPU and GPU timings scale close to linearly, approximately doubling as S doubles, as expected. The speedup factor is approximately 11X on average, which is to say that the GPU computations are about 11 times faster than the CPU computations. For lower values of $\log_2 S$ the speedup is a little less, because the overhead of memory transfers between host and device memories is spread out over less computational time.

Table 3 contains similar timings for the auction model. Again, the timings scale close to linearly in S. For this model, the speedup from moving to GPU computing is considerably greater, averaging about 40X for the sizes of simulations considered, and reaching a little more than 55X in the best case. The greater speedup for the auction model compared to the MA model is likely due to the fact that the auction model involves considerably more complex computations, but requires considerably less memory usage, as it involves a smaller sample size and a lower dimensional auxiliary statistic. Again, the speedup is better for larger $\log_2 S$, so the time spent doing calculations is larger compared to the time for memory transfers.

The speedups are very satisfactory, in our opinion. For the sort of computations required for IL estimation, a single graphics card that costs around \$230 obtains performance equivalent to at least 9 and up to 55 CPU cores. A typical cluster would consist of rackmount servers, and a fairly typical rackmount server may contain two quad core CPUs, for a total of 8 cores. The cost of such a server is roughly 10 times that of the GPU card. For the auction model, one would need four such servers to equal the performance of the single GPU card, for a total cost of roughly 40 times that of the GPU card. The maintenance cost of such a cluster, as well as the cost of powering and cooling it is considerable, as is the noise that it makes. In contrast, the GPU device can be installed in a single cheap, quiet, energy efficient desktop computer, and one only needs maintain a single computer.

These results use a single GPU device, and a single query point. When multiple query points are used, which is the case when doing Monte Carlo, or when running multiple MCMC chains, the nearest neighbors part of the computations becomes more demanding, and it becomes beneficial to seek further possibilities for parallelization. It is straightforward to spread the computations over several GPU devices, using OpenMP. The code that accompanies this paper detects and uses multiple GPU devices, if available.

4. Econometric results

In this section we extend the results of Creel and Kristensen (2011) by exploring the effect of increasing the number of simulations, S, and also systematically exploring the choice of the number of neighbors, K. In Creel and Kristensen (2011), the number of simulations used in various examples is between 10^6 and 10^7 , and the paper contains no investigation of the effect of the choice of S on the performance of the estimators. Also, in that paper, a simple rule setting $K = 1.5 \times S^{0.25}$ is used to select the number of neighbors for the nonparametric fit. With GPU computing, it is much quicker do Monte Carlo work, which facilitates more careful study of the performance of the IL estimators in relation to the tuning of the nonparametric fitting methods used.

We present results for the MA model of Section 2.2.2. The findings are very similar for all 7 design points, so we focus on the case of $\psi = 0.9$, so as to present fewer tables. First, we generate 5000 Monte Carlo replications of Z_n. Then we generate S replications of Z_n^s, and find the 300 nearest neighbors to each of the 5000 Monte Carlo replications of Z_n. This information is saved. This process is done for the values $\log_2 S \in \{12, 14, ..., 24\}$. Then we can compute the SBIL estimator using any number of neighbors up to 300, which is the most that were saved.

Table 4 presents root mean squared error (RMSE) for estimation of ψ, as a function of $\log_2 S$ and K. In this table, the minimum RMSE values tend to lie on the "main diagonal". We can see that for a given $\log_2 S$, RMSE has a U shape as a function of K, first declining to a minimum, then rising. For the larger values of $\log_2 S$, we do not observe the rise, as we have not computed enough neighbors. This shape is expected. For a given $\log_2 S$, when K is small, bias is small, but the variance is large, because we are averaging few neighbors. When K is large, parameter values far from the true value will be included among the neighbors, provoking a large bias. Likewise, for a given K, there is a value of $\log_2 S$ that minimizes RMSE, which is seen clearly in the first rows of the Table. When $\log_2 S$ is too small for a given K, the pool of potential neighbors is too small, and we are forced to include parameter values that are far from the true value, provoking excess bias. When $\log_2 S$ is too large, there will be relatively many Z_s that are realized in the tails of their conditional distributions given θ_s, and their inclusion among the neighbors provokes an increase in the variance. The table confirms the result from theory that K should be an increasing function of S. We note that the rule relating K and S used in Creel and Kristensen (2011) leads to too small of a value of K. For example, when $\log_2 S = 20$, $S = 1,048,576$, and the rule $K = 1.5 \times S^{0.25}$gives $K = 48$. The RMSE reported in Creel and Kristensen (2011) using this value of K is 0.042. The minimum RMSE value in Table 4 is 0.040, with the corresponding $K \geq 150$. This indicates that there may be scope for improving results by more careful selection of S and K. However, the difference is quite small, and we observe in the Table that RMSE is fairly constant over wide ranges of S and K, so excessive effort devoted to searching is not warranted. Also, moderate values of S can give good results. It is to be kept in mind, however, that the dimension of the parameter vector is only 2 in this example, and that higher dimensions will require larger simulated samples. When this is the case, the speedup from GPU computing will become even more attractive.

5. Conclusions

Within econometrics, many procedures have a data parallel structure that requires minimal logical branching and which have a high computational intensity. This paper has shown that GPU computing can obtain good speedups for this type of work. Likelihood functions for Markovian data, Monte Carlo, simple bootstrapping, nonparametric fitting methods based on local averaging, and similar problems have structures that are amenable to GPU computing. This is certainly not to suggest that GPU computing is a good solution for all computing problems in econometrics. Models for non-Markovian data normally require sequential computations, which do not fit the data parallel paradigm. Methods such as MCMC or simulated annealing are essentially sequential, and even if one considers ensemble versions that can be parallelized with relative ease, the problem of thread divergence can arise, as different parts of the ensemble take different execution paths. This could limit the gains from moving such computations to the GPU, and given the greater complexity of programming for GPUs, such problems are probably best tackled using CPU-based solutions, at least at present.

GPU computing, when it is applicable, has many advantages over alternative methods of parallelization, such as clustering machines for CPU-based computation, in that the hardware, energy and maintenance cost is much lower for GPU computing. It is true that programming for GPU computing is more complex than is standard programming for CPU computing. In addition to the usual challenges of thinking in terms of parallel computing, the available libraries of code to accomplish needed tasks are more limited. For example, we had to program the OLS estimation in the CUDA kernel from scratch. However, there is no doubt that this problem will become less severe in the future, as more libraries become available. Our code for OLS fitting is now available, and can be used in other applications of GPU computing. It is also possible to port only some parts of a computational problem to the GPU, as we have done with the computation of the likelihood function. A large and complex CPU-based software body can benefit from selective use of GPU computing by porting only computational bottlenecks that are well adapted to the data parallel and branch-free paradigm that suits GPU computing. We reiterate that the code that accompanies this paper is available from the authors[4]. It is extensively commented, with instructions for its use. We hope that this code can

[4]The code is archived at pareto.uab.es/mcreel/GPU_paper_code_release_v1.zip

help other econometricians to learn to use GPU computing for their research interests.

Tables

Table 1: Time to compute 2000 evaluations of Poisson likelihood function.

$log_2(S)$	8	10	12	14	16
CPU (ms)	114	210	572	2086	7037
GPU (ms)	74	75	85	148	393
Speedup (CPU/GPU)	1.54	2.80	6.73	14.10	17.90

Table 2: MA(1) model, time to simulate and find 50 neighbors to Z_n.

$log_2(S)$	17	18	19	20
CPU (s)	29.80	57.26	113.45	221.48
GPU (s)	3.15	5.41	9.93	19.07
Speedup (CPU/GPU)	9.46	10.58	11.42	11.61

Table 3: Auction model, time to simulate and find 50 neighbors to Z_n.

$log_2(S)$	17	18	19	20
CPU (s)	27.98	53.12	104.36	205.61
GPU (s)	1.23	1.57	2.29	3.71
Speedup (CPU/GPU)	22.75	33.83	45.57	55.42

Table 4: MA(1) model, estimation of ψ, RMSE as a function of $log_2 S$ and K

	$log_2(S)$						
K	12	14	16	18	20	22	24
10	0.040	0.046	0.046	0.044	0.044	0.044	0.044
20	0.040	0.042	0.043	0.042	0.042	0.042	0.043
30	0.040	0.041	0.042	0.041	0.042	0.042	0.042
40	0.042	0.041	0.041	0.041	0.041	0.041	0.042
50	0.045	0.040	0.041	0.041	0.041	0.041	0.042
100	0.058	0.041	0.040	0.040	0.041	0.041	0.041
150	0.066	0.042	0.040	0.040	0.040	0.041	0.041
200	0.075	0.044	0.040	0.040	0.040	0.040	0.041
250	0.089	0.046	0.041	0.040	0.040	0.040	0.041
300	0.107	0.048	0.041	0.040	0.040	0.040	0.041

References

[1] Aldrich, E., Fernández-Villaverde, J., Gallant, A.R., and Rubio-Ramírez, J.F., 2011, "Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors," *Journal of Economic Dynamics and Control*, 35, 386–393.

[2] Beaumont, M., W. Zhang and D. Balding, 2002, "Approximate Bayesian computation in population genetics", *Genetics*, 162, 2025-2035.

[3] Chernozhukov, V. and H. Hong, 2003, "An MCMC approach to classical estimation," *Journal of Econometrics,* 115, 293-346.

[4] Creel, M., 2005, "User-friendly parallel computations with econometric examples", *Computational Economics*, 26, 107-128.

[5] Creel, M. and D. Kristensen, 2011, "Indirect likelihood inference", Barcelona GSE Working Paper 558, `http://research.barcelonagse.eu/tmp/working_papers/558.pdf`.

[6] Durham, G. and J. Geweke, 2011, "Massively Parallel Sequential Monte Carlo for Bayesian Inference", `http://ssrn.com/abstract=1964731`.

[7] Garcia, V., E. Debreuve and M. Barlaud, 2008, "Fast k nearest neighbor search using GPU", in *Proceedings of the CVPR Workshop on Computer Vision on GPU*, Anchorage, Alaska, USA.

[8] Garcia, V., 2008, Ph.D. Thesis: *Suivi d'objets d'intérêt dans une séquence d'images : des points saillants aux mesures statistiques,* Université de Nice - Sophia Antipolis, Sophia Antipolis, France.

[9] Hall, R. 1970, "The calculation of ordinary least squares estimates", `http://stanford.academia.edu/RobertEHall/Papers/63352/The_Calculation_of_Ordinary_Least_Squares_Estimates`

[10] Li, S., 2011, *Three essays on econometrics: asymmetric exponential power distribution, econometric computation, and multifactor model*, PhD. dissertation in Economics, Rutgers, The State University of New Jersey.

[11] Li, T., 2010, "Indirect inference in structural econometric models," *Journal of Econometrics,* 157, 120-128.

[12] Li, Q. and J. Racine, 2007, *Nonparametric Econometrics: Theory and Practice.* Princeton: Princeton University Press.

[13] Liepe, J., C. Barnes, E. Cule, K. Erguler, P. Kirk, T. Ton, and M. Stumpf, 2010, "ABC-SysBio—approximate Bayesian computation in Python with GPU support ", *Bioinformatics*, 26, 1797–1799.

[14] Marjoram, P., J. Molitor, V. Plagnol and S. Tavaré, 2003, "Markov chain Monte Carlo without likelihoods", *Proceedings of the National Academy of Sciences, USA*, 100, 15324-15328.

[15] Morozov, S. and S. Mathur, 2012, Massively parallel computation using graphics processors with application to optimal experimentation in dynamic control, *Computational Economics*, 40, 151-182.

[16] Phillips, E., 2010, "Cuda accelerated LINPACK on clusters", `http://www.nvidia.com/content/PDF/sc_2010/theater/Phillips_SC10.pdf`

[17] Racine, Jeff (2002) Parallel distributed kernel estimation, *Computational Statistics & Data Analysis*, 40, 293-302.

[18] Sisson, S., Y. Fan and M. Tanaka, 2007, "Sequential Monte Carlo without likelihoods", *Proceedings of the National Academy of Science, USA*, 104, 1760-1765.

[19] Suchard, M, Q. Wang, C. Chan, J. Frelinger, A. Cron, and M. West, 2010, "Understanding GPU Programming for Statistical Computation: Studies in Massively Parallel Massive Mixtures ", *Journal of Computational and Graphical Statistics*, 19, 419–438.

[20] Swann, C.A., 2002, "Maximum likelihood estimation using parallel computing: an introduction to MPI", *Computational Economics*, 19, 145-178.

[21] Tavaré, S., D. Balding, R. Griffiths and P. Donnelly, 1997, "Inferring coalescence times from DNA sequence data", *Genetics*, 145, 505-518.