ECONOMETRICS ON GPUS

MICHAEL CREEL AND MOHAMMAD ZUBAIR

Abstract. A graphical processing unit (GPU) is a hardware device normally used to manipulate computer memory for the display of images. GPU computing, also known as general purpose 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 ability of a GPU to render rapidly changing complex images depends on the ability of the GPU to perform floating point operations related to rotations, scaling, shading, etc. Such operations exhibit data parallelism where different computational units operating simultaneously on independent data. GPUs have become extremely powerful for this sort of work. A number of scientific applications have a data parallel structure, and realizing this, vendors and other parties have developed programming frameworks to make it relatively easy to conduct general purpose computation using GPUs. In this paper, we show how some problems in econometrics have the data parallel structure that allows for successful use of GPU computing. We find speedups from 9.5 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. The paper is accompanied by example code that illustrates how econometric computations can be done using one or a number of GPUs. The main intent of the paper is to contribute to the limited number of examples of GPU computing in economics and econometrics, and to provide example code that can be adapted with relative ease to other problems.

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. Largely due to the willingness of computer gamers to pay for realistic, fast-paced 3D action, 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 great 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 recently, thousands of cores.

GPU computing, also known as general purpose 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

This work was supported by grants MICINN-ECC2009-11857 and SGR2009-578.
many cores of the GPU to accelerate computations by offloading part of the work
of the CPU to the GPU. Certain computations exhibit data-level parallelism and
some of these are suitable for programming on GPUs. Like other traditional parallel
computing hardware, a GPU is suitable for computations that have high arithmetic
intensity, where arithmetic intensity is the number of operations per byte of memory
transferred (Patterson and Hennessy, 2012). In a data parallel environment it is
possible for a core to execute a different operation on its data, depending on what
data it has. To support efficient execution of hundreds of threads running different
instructions, 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 thread divergence are not necessarily good candidates
for porting to the GPU. Within econometrics, many problems of interest fit the data
parallel paradigm, and it is often possible to avoid thread divergence with careful
programming. Creel (2005) discusses using the message passing interface (MPI) to
accelerate Monte Carlo work, bootstrapping, and kernel regression. These problems
are “embarrassingly parallel”, which means that they are almost fully data parallel,
with threads taking a single execution path. As such, parallelization is both easy
to implement and gives an excellent speedup. Such problems are prime candidates
for porting to the GPU.

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 comput-
ing architecture and software development kit, which allowed programming for
the GPU 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
reported on that page range from 1 to 3 orders of magnitude, compared to comput-
ations done using a single CPU thread. CUDA and the associated programming
environment generates code that runs only on hardware devices from Nvidia. The
OpenCL\(^1\) 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 pro-
gramming and mixed CPU/GPU programming using hardware devices from a num-
ero 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 programming 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

\(^1\)http://www.khronos.org/opencl/
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 Gflops/s per watt. At the time of this writing (March, 2012), the GTX460 is available for roughly $150, while the GTX560 Ti 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 cheap potential 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 (2011) 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), and 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 cores, 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.

We use as an example 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 achieve a speedup factor from 11 to 34 times the speed of computations done using a single CPU thread. We expect that similar speedups would apply to many other problems in econometric estimation.

2. Example: indirect likelihood inference

This section reviews the estimators that form the example. Indirect likelihood inference (Creel and Kristensen, 2011) is a 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 this 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.

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 $Y_n = (y_1, \ldots, y_n)$ from the model, we make inference on $\theta$ through a $d$-dimensional statistic of the sample, $Z_n = Z_n (Y_n) \in \mathbb{R}^q$. We can think of $Y_n$ as a (random) mapping taking a parameter value into the corresponding observed sample, $Y_n = Y_n (\theta)$. This in turn implies that the statistic also implicitly is a function of $\theta$ through the data, and write

$$Z_n (\theta) \equiv Z_n (Y_n (\theta)).$$

The observed statistic is a realization of this random mapping evaluated at the true parameter value which we denote $\theta_0$, $Z_n = Z_n (\theta_0)$. 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 estimator maximizes the indirect likelihood defined through $Z_n$:

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

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 $k$, 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 \( \theta \) given \( Z_n \) defined as

\[
\hat{\theta}_{BIL} = E(\theta | Z_n) = \int_{\Theta} \theta f_n(\theta|Z_n) d\theta,
\]

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

\[
f_n(\theta|Z_n) := \frac{f_n(Z_n, \theta)}{f_n(Z_n)} = \frac{f_n(Z_n|\theta)\pi(\theta)}{\int_{\Theta} f_n(Z_n|\theta)\pi(\theta) d\theta}
\]

for some pseudo-prior density \( \pi(\theta) \) on the parameter space \( \Theta \). We refer to this particular estimator as the Bayesian indirect likelihood (BIL) estimator. Now we note that 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, as follows:

2.1. SMIL. We have to be able to compute \( f_n(Z_n|\theta) \) at any given trial value \( \theta \). Since the model is simulable and the mapping \( Z_n(\theta) = Z_n(Y_n(\theta)) \) is known (as chosen by the econometrician), we propose to estimate the density using nearest neighbor methods: Draw \( S \) independent samples, \( Y_n^s(\theta) \) for \( s = 1, \ldots, S \), from the model evaluated at the trial value \( \theta \), compute the associated statistic, \( Z_n^s(\theta) = Z_n(Y_n^s(\theta)) \). Let the set of \( S \) such simulations be \( Z_S(\theta) = \{Z_n^s(\theta), s = 1, \ldots, S\} \). Note that the elements of \( Z_S \) are independently and identically distributed. Let \( K \) be a positive integer, and let \( c_0 \) be the volume of the unit ball in \( \mathbb{R}^q \). Finally, let \( d_K(Z_n, Z_S(\theta)) \) be the Euclidean distance between \( Z_n \) and the \( K^{th} \) nearest neighbor to \( Z_n \) in \( Z_S(\theta) \). A simple K-nearest neighbors density estimator (see Li and Racine, 2007, equation 14.2) is

\[
\hat{f}_n, S(Z_n|\theta) = \frac{K}{c_0 S[d_K(Z_n, Z_S(\theta))]^q},
\]

One can define more complicated and perhaps better nearest neighbors density estimators using weighting functions, but this simple possibility captures the essence of the idea and is sufficient for the points we address in this paper. When \( K \) is chosen to be an appropriately slowly growing function of the number of simulations, \( S \), this density estimator is consistent for the true density \( f_n(\theta|Z_n) \) as \( S \) goes to infinite (add reference). This means that \( \hat{f}_n, S(Z_n|\theta) \) can be made very accurate for \( f_n(\theta|Z_n) \) by setting \( S \) sufficiently large (note that \( n \) is fixed). The last step is to embed the approximated density inside (2.1), and use an optimization algorithm to obtain an estimator. This yields a simulated MIL (SMIL) estimator:

\[
\hat{\theta}_{SMIL} = \arg \max_{\theta \in \Theta} \log \hat{f}_n, S(Z_n|\theta).
\]

Note that for given \( S \) and \( K \), an equivalent definition of the estimator is

\[
\hat{\theta}_{SMIL} = \arg \min_{\theta \in \Theta} d_K(Z_n, Z_S(\theta))
\]

(see equation 2.4) which means that we can simply minimize the distance between \( Z_n \) and the \( K^{th} \) neighbor in the set \( Z_S(\theta) \). Creed and Kristensen (2011) discusses the properties of this estimator and the conditions under which it behaves asymptotically equivalently to the infeasible MIL estimator.

To compute the SMIL estimator, one needs to generate a number of simulations \( Z_S(\theta) \) at the different trial values \( \theta \) that the optimization algorithm uses along the path to the minimum, and one needs to find the \( K^{th} \) nearest neighbor to \( Z_n \) in each of these large sets of simulated statistics. The SMIL objective function has
multiple local maxima, due to the use of nonparametric methods. For this reason, a global minimizer such as simulated annealing must be used. Because a large number of objective function evaluations must be done, and each of these entails a simulation and a nearest neighbors nonparametric estimation, the SMIL estimator is computationally demanding.

2.2. SBIL. The BIL defined above (equation 2.2) is the 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 \( \theta_s \), \( s = 1, \ldots, S \), 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 \( Z^s_n = Z_n(Y_n(\theta^s)) \), \( s = 1, \ldots, S \). Now let \( Z_S = \{ Z^s_n, s = 1, 2, \ldots, S \} \) be the set of the simulated statistics. This is exactly as before, except that each element of the set is generated at a different random draw of \( \theta \). We can obtain a simulated version of the BIL (SBIL) through nonparametric regression techniques. One such is a simple \( K \) nearest neighbors regression estimator (see Li and Racine, 2007, Ch. 14),

\[
\hat{\theta}_{SBIL} = \frac{1}{K} \sum_{s=1}^{S} \theta^s \mathbf{1}(\|Z^s_n - Z_n\| \leq d_K(Z_n, Z_S)),
\]

where \( \mathbf{1}(\cdot) \) is the indicator function and \( d_K(Z_n, Z_S) \) is the Euclidean distance between \( Z_n \) and the \( K \)th closest element of \( Z_S \) to \( Z_n \). Simply put, this estimator is the average of the \( K \) values of \( \theta^s \) that lead to the \( K \) closest neighbors to \( Z_n \). Again, 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.

2.3. SBIL-MCMC. Applying Chernozhukov and Hong’s ideas when the original estimator is the MIL leads to the BIL estimator, which they propose to compute using MCMC. The ABC literature has also proposed to use MCMC instead of the simple sampling from the prior that characterizes the SBIL estimator, see Marjoram et al. (2003). Consider the Metropolis-Hastings MCMC algorithm. The prior density is \( \pi(\theta) \), as above. Let \( g(\theta' ; \theta) \) be the proposal density, which generates a new trial parameter value \( \theta' \) given the most recently accepted parameter value \( \theta \). A proposal will be accepted if

\[
\frac{f(\theta'|Z_n) \, g(\theta;\theta')}{f(\theta|Z_n) \, g(\theta';\theta)} > \alpha
\]

where \( \alpha \) is a \( U(0,1) \) random variate. Note that the ratio of posteriors is equal to the ratio of likelihoods times the ratio of priors:

\[
\frac{f(\theta'|Z_n)}{f(\theta|Z_n)} = \frac{f(Z_n|\theta') \, \pi(\theta')}{f(Z_n|\theta) \, \pi(\theta)}
\]

From this, we see that the information needed to determine if a proposal is accepted or rejected is the prior, the proposal density, and the likelihood function \( f(Z_n|\theta) \).
The likelihood function is normally not available, but we can use the consistent (as $S$ increases) estimator of equation 2.4 in its place. If this is done, then a proposal will be accepted if

$$\left( \frac{\hat{f}_{n,S}(Z_n|\theta')}{\hat{f}_{n,S}(Z_n|\theta)} \right) \left( \frac{g(\theta'; \theta')}{g(\theta; \theta)} \right) \left( \frac{\pi(\theta')}{\pi(\theta)} \right) > \alpha$$

Note that the first term can be expressed in terms of the distances from $Z_n$ to the $K^{th}$ nearest neighbor of the simulated values:

$$\hat{f}_{n,S}(Z_n|\theta') = \left( \frac{d_K(Z_n, Z_S(\theta))}{d_K(Z_n, Z_S(\theta'))} \right)^q$$

Thus, the same information (the distances or the likelihood function values) that is needed to compute the SMIL estimator can be used to compute a simulated BIL (SBIL) estimator, using MCMC methods. To compute the SBIL estimator using MCMC, one must run the chain for some large number of iterations, so the computational demand is high, in common with the SMIL estimator. The likelihood function must be evaluated many times, and each evaluation entails a simulation and a nearest neighbors nonparametric fit.

2.4. **Summary.** Comparing the SMIL, SBIL, and SBIL-MCMC estimators, the SBIL estimator requires generating a single large set of simulations, each done at a value of the parameter drawn from the prior. From this large body of simulated statistics, nearest neighbors to the observed sample statistic $Z_n$ are computed, and the estimator is simply the mean of these neighbors. The SMIL and SBIL-MCMC estimators require a number of simulations and nearest neighbors computations to be done sequentially, where each simulation in a set is done at the same parameter value, and the algorithm determines how to update the parameter values. Because of this difference, one can contemplate using a much larger $S$ for the SBIL estimator than what would be feasible for the SMIL or SBIL-MCMC estimators. It is also worth noting that the simple simulation and nearest neighbors computations behind the SBIL estimator have no numeric instabilities and require no tuning, while successful use of MCMC can require skill and care (Korniltsina and Nekipelov, 2009). It is easy to implement SBIL for a number of $Z_n$ simultaneously, which greatly facilitates Monte Carlo study of the properties of the estimator. The same large simulation can be used for each of a number of $Z_n$, we only need to keep track of the different neighbors to each of them.

The SMIL, SBIL, and SBIL-MCMC estimators discussed here all involve running relatively long (for accuracy) simulations, and nonparametric fitting done using nearest neighbors. A long simulation and the determination of nearest neighbors are both problems that are computationally demanding. Showing that these tasks can be accelerated by using GPU computing will benefit all three of the estimators discussed in this section. In the following, we will show how simulation and nearest neighbors computations can be accelerated, and then we will give some results, focusing on the SBIL estimator.
3. TWO EXAMPLE MODELS

3.1. MA model. A first model, chosen for its simplicity, is the MA(1) model

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

We use a sample sizes of \(n=200\) observations. The parameter \(\psi\) 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 \(\sigma\) 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 statistic \(Z_n\) is the vector of estimated parameters \((\rho_0, \rho_1, ..., \rho_P, \sigma^2)\) of an AR(\(P\)) model \(y_t = \rho_0 + \sum_{p=1}^{P} \rho_p y_{t-p} + \epsilon_t\), 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 \(\theta\) is 2, so we have considerable overidentification.

3.2. 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 \(\theta_0\) and \(\theta_1\). 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 \(Z_n = (\hat{\alpha}, \hat{\beta}, \log \hat{\sigma})\). 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.
4. The Code and benchmarking

In this section we describe the CUDA code and give benchmarking results that illustrate the speedup in comparison to CPU-based code. The code that accompanies the paper is extensively commented and documented, so here we address general issues and explain the overall strategy used to port the code to the GPU.

4.1. The code. We provide Matlab/Octave code to estimate the MA and auction models by SBIL, SMIL and SBIL-MCMC. The Matlab/Octave code is very simple, and interfaces with CUDA through text files for input/output. The CUDA code for SMIL and SBIL-MCMC is identical, taking a trial value of $\theta$ and returning the nearest neighbors to $Z_n$, plus the distances from each neighbor to $Z_n$. These distances can be used to implement SMIL or SBIL-MCMC, following equations 2.6 and 2.8. The CUDA code for SBIL is very similar to the SMIL and SBIL-MCMC code, except that no $\theta$ value is given as an input, because the trial $\theta$ values are drawn from the prior distribution, as described above. Because the two versions of the code are very similar, here we describe only the SBIL code.

We first give a high-level description of the computation required for SBIL model, and then look at issues on implementing on one or more GPU devices on the system. There are three major steps in implementing the SBIL mode.

- **Step 1.** The real sample data is used to compute the statistic $Z_n$. If one is doing Monte Carlo, a set of $nZn$ samples are generated to generate $nZn$ replications, $Z_{n}^{s}$, $j = 1, 2, ..., nZn$. $nZn$ is 1 when real sample data is used, or of the order of thousands when doing Monte Carlo. This part of the code is implemented in Matlab/Octave.

- **Step 2.** Compute simulated statistics $Z_{S} = \{Z_{n}^{s}, s = 1, 2, ..., S\}$ where $S$ is of the order of millions. This is done using CUDA.

- **Step 3.** For each of the $Z_{n}^{s}$, $j = 1, 2, ..., nZn$, find the $K$ nearest neighbors in $Z_{S} = \{Z_{n}^{s}, s = 1, 2, ..., S\}$, where $k$ is of the order of 10s or 100s. Record the $\theta$ that generated each of the neighbors. This is done using CUDA.

- **Step 4.** Once the neighbors are found and the $\theta$ that generated them are recorded, the SBIL estimator can easily be computed as a mean or median. This is done using Matlab/Octave.

We focus on steps 2 and 3, as they are the most computationally intensive and involve methods that are unlikely to be familiar to many readers. We first consider values $S$ and $nZn$ such that all the required data fits in the device memory of GPU. We found these values to be $S = 2^{19}$ and $nZn = 1000$ on a Tesla M2050 GPU device, and $S = 2^{17}$ and $nZn = 1000$ on a GTX560Ti device. The code gives an out of memory message one attempts to use values that are too large, so appropriate values for a given device are easily discovered.

4.1.1. Computation of the simulated $Z_{n}^{s}$. Computation of $Z_{n}^{s}$ requires first sampling $\theta$ from the prior distribution $\pi(\theta)$, which involves generation of uniform random numbers. Then for a given $\theta$, a sample of size $n$ is generated, which in our examples involves uniform, normal, and exponential random draws. The simplest way to parallelize the whole computation on a GPU is to assign a thread for computing one or more of the $Z_{n}^{s}$. The random numbers required by each thread are generated using CURAND, a CUDA based library for generating random numbers. Using this library there are two ways to generate random numbers. The first is
to make a call to the CURAND API on the host-side code to generate random numbers. The CURAND API uses GPU device to generate random numbers which are stored in the global memory of the device for the application kernel to use later. The second way is to call the device based CURAND API in the user defined kernel to generate random numbers without requiring the random numbers to be written to and then read from the global device memory. We use both methods in our implementation. The first method is used for generating the trial parameter values $\theta^*$ from the prior distribution $\pi(\theta)$. The second method is used for generating the $O(n)$ random numbers that are the random shocks of the simulated sample data. This involves many more draws, so these random numbers are generated on fly in the kernel, to avoid using excess memory. Both the MA and auction example use an OLS fit as the auxiliary statistic, and at the kernel level there are no standard libraries available that support ordinary least square computation. We wrote code for the OLS computation in the CUDA kernel using an algorithm based on Cholesky factorization, using Hall (1970) as a guide. This part of our code may be useful for econometric applications using CUDA that are not related to the IL estimators used as examples in this paper.

4.1.2. Computation of $k$ nearest neighbors. For this computation 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.

4.1.3. Large Problem Size. To address large size problem, where $S > 2^{19}$ (or whatever limit the device supports) and/or $nZn > 1000$, we iterate simulation and nearest neighbors search. First we generate $2^{19}$ simulations of $Z_n$ (target points) and find the neighbors and distances to the first 1000 query points of the $nZn$ points $Z_n$. Then we loop over blocks of query points until we have found the $K$ nearest neighbors and distances for all of the $2^{19}$ query points. Then we repeat this process for the next block of $2^{19}$ simulated target points. Next we merge the two sets of neighbors, by comparing distances, to select the $K$ nearest neighbors out of the two sets of $K$ neighbors, for each of the $nZn$ query points. This entire process is inside a loop, which runs until $S$ target points have been generated. It is to be noted that this process can easily be made to use multiple GPU devices. Our code detects and uses multiple GPU devices, if present.

4.2. Benchmarking. For all three estimators, 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 GPU computations are 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 card is installed in the same desktop computer that is used for the CPU timings.

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, $Z_S$, 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 1 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 LS 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 2 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 LS, 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.

5. 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, a simple rule setting $K = 1.5S^{0.25}$ is used to select the number of neighbors for the nonparametric fit. With GPU computing, it is much quicker to 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 3.1. The findings are very similar for all 7 design points, so we focus on the case of $\psi = 0.9$ here, so as to present fewer tables. First, we generate 5000 Monte Carlo replications of $Z_n$. Then we generate $S$ replications of $Z_n^a$, 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 $S \in \{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}, 2^{24}\}$. Then we can compute the SBIL estimator using any number of neighbors up to 300, which is the most that were saved.

Table 3 presents root mean squared error (RMSE) for estimation of $\psi$, as a function of $LS$ and $K$. In this table, the minimum RMSE values tend to lie on the “main diagonal”. We can see that for a given $LS$, RMSE has a U shape as a function of $K$, first declining to a minimum, then rising. For the larger values of $LS$, we do not observe the rise, as we have not computed enough neighbors. This shape is expected. For a given $LS$, 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 $LS$ that minimizes RMSE, which is seen clearly in the first rows of the Table. When $LS$ 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 $LS$ is too large, there will be relatively many $Z_n$ that are realized in the tails of their conditional distributions given $\theta_s$, 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 $LS = 20$, $S = 1,048,576$, and the rule $K = 1.5S^{0.25}$ 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 3 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 $K$. The good
news is that RMSE is fairly constant over $S$ 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.

6. Conclusions

Within econometrics, many procedures have a data parallel structure. Monte Carlo work, bootstrapping, nonparametric fitting using kernel or nearest neighbors methods are well known examples (Creel, 2005). Objective functions that define econometric estimators also often fit the data parallel paradigm. This paper has shown that GPU computing can obtain impressive speedups for this type of work, using the example of indirect likelihood estimators. When a problem can be broken into many independent threads, GPU computing can accelerate computations by a very considerable margin.

GPU computing 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. There are also some important limitations to GPU computing. In many problems of interest, it is not possible to organize computations into many independent threads. Problems with much logical branching, or inherently sequential problems, are not good candidates for GPU computing. Also, 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. The code that accompanies this paper is available at addurlhere. It is extensively commented, with instructions for its use, and it allows replicating the results reported in this paper, as well as to compute the SBIL estimator for the two example models. We hope that this code can help other econometricians to learn to use GPU computing for their research interests.
Table 1. MA model, time to simulate and find 50 neighbors to $Z_n$.

<table>
<thead>
<tr>
<th>$\log_2(S)$</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (s)</td>
<td>29.80</td>
<td>57.26</td>
<td>113.45</td>
<td>221.48</td>
</tr>
<tr>
<td>GPU (s)</td>
<td>3.15</td>
<td>5.41</td>
<td>9.93</td>
<td>19.07</td>
</tr>
<tr>
<td>Speedup (CPU/GPU)</td>
<td>9.46</td>
<td>10.58</td>
<td>11.42</td>
<td>11.61</td>
</tr>
</tbody>
</table>

Table 2. Auction model, time to simulate and find 50 neighbors to $Z_n$.

<table>
<thead>
<tr>
<th>$\log_2(S)$</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (s)</td>
<td>27.98</td>
<td>53.12</td>
<td>104.36</td>
<td>205.61</td>
</tr>
<tr>
<td>GPU (s)</td>
<td>1.23</td>
<td>1.57</td>
<td>2.29</td>
<td>3.71</td>
</tr>
<tr>
<td>Speedup (CPU/GPU)</td>
<td>22.75</td>
<td>33.83</td>
<td>45.57</td>
<td>55.42</td>
</tr>
</tbody>
</table>

Table 3. MA model, estimation of $\psi$, RMSE as a function of $\log_2 S$ and $K$

<table>
<thead>
<tr>
<th>$\log_2(S)$</th>
<th>$K$</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
<th>22</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.040</td>
<td>0.046</td>
<td>0.046</td>
<td>0.044</td>
<td>0.044</td>
<td>0.044</td>
<td>0.044</td>
<td>0.044</td>
</tr>
<tr>
<td>20</td>
<td>0.040</td>
<td>0.042</td>
<td>0.043</td>
<td>0.042</td>
<td>0.042</td>
<td>0.042</td>
<td>0.042</td>
<td>0.043</td>
</tr>
<tr>
<td>30</td>
<td>0.040</td>
<td>0.041</td>
<td>0.042</td>
<td>0.041</td>
<td>0.042</td>
<td>0.042</td>
<td>0.042</td>
<td>0.042</td>
</tr>
<tr>
<td>40</td>
<td>0.042</td>
<td>0.042</td>
<td>0.041</td>
<td>0.041</td>
<td>0.041</td>
<td>0.041</td>
<td>0.042</td>
<td>0.042</td>
</tr>
<tr>
<td>50</td>
<td>0.045</td>
<td>0.040</td>
<td>0.041</td>
<td>0.041</td>
<td>0.041</td>
<td>0.041</td>
<td>0.041</td>
<td>0.042</td>
</tr>
<tr>
<td>100</td>
<td>0.058</td>
<td>0.041</td>
<td>0.040</td>
<td>0.040</td>
<td>0.041</td>
<td>0.041</td>
<td>0.041</td>
<td>0.041</td>
</tr>
<tr>
<td>150</td>
<td>0.066</td>
<td>0.042</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.041</td>
</tr>
<tr>
<td>200</td>
<td>0.075</td>
<td>0.044</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.041</td>
</tr>
<tr>
<td>250</td>
<td>0.089</td>
<td>0.046</td>
<td>0.041</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.041</td>
</tr>
<tr>
<td>300</td>
<td>0.107</td>
<td>0.048</td>
<td>0.041</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.040</td>
<td>0.041</td>
</tr>
</tbody>
</table>

Tables
REFERENCES


