High Performance Implementation of an Econometrics and Financial Application on GPUs

Michael Creel
Universitat Autònoma de Barcelona
Barcelona Graduate School of Economics
Barcelona, Spain
Email: michael.creel@uab.es

Mohammad Zubair
Department of Computer Science
Old Dominion University
Norfolk, Virginia
Email: zubair@cs.odu.edu

Abstract—In this paper, we describe a GPU based implementation for an estimator based on an indirect likelihood inference method. This method relies on simulations from a model and on nonparametric density or regression function computations. The estimation application arises in various domains such as econometrics and finance, when the model is fully specified, but too complex for estimation by maximum likelihood. We implemented the estimator on a machine with two 2.67GHz Intel Xeon X5650 processors and four NVIDIA M2090 GPU devices. We optimized the GPU code by efficient use of shared memory and registers available on the GPU devices. We compared the optimized GPU code performance with a C based sequential version of the code that was executed on the host machine. We observed a speed up factor of up to 242 with four GPU devices.

I. INTRODUCTION

Power consumption and heating constraints are limiting the instruction level parallelism for improving processor performance. We have reached a stage in the evolution of computer architecture where the only way to get high performance for an application is to program it for multiple cores. This along with the fact that a cost of a processor with manycores has gone drastically down is making it very attractive to port applications from different discipline onto these platforms. With the advent of the NVIDIA GPU architecture, one can purchase a laptop for under $1K with manycores. Consequently, parallel computing which in the past was mostly available to scientific applications can now be applied to other disciplines such as econometrics and financial computing. Additionally, compared to other multicore processors, NVIDIA GPUs consume less energy per flop.

Though we have access to cheap multiple cores, the software is still lagging behind the hardware in utilizing these cores on a processor. Applications need to be explicitly coded to exploit multiple cores. In general, parallel programming is difficult as it requires a change in programming philosophy and retraining. The biggest challenge today is to be able to design and implement high performance algorithms that take advantage of multiple cores on the target hardware. The availability of high level tools has made it easy to write a parallel program, however, the difficulty still remains in writing a program that runs efficiently and its performance is close to the theoretical peak performance of the hardware. NVIDIA’s Compute Unified Device Architecture (CUDA) provides an easy hybrid programming model for CPUs and GPUs. However, the problem of partitioning the task between cores of CPU and cores of GPU along with coordination of these subtasks is still complex to implement. Recent release of CUDA toolkit 4.0 added support to ease some of these issues. Additionally debugging a program on a GPU and particularly when working with multiple GPU devices on a system can be challenging because of lack of integrated debugging tools. Recently NVIDIA along with Cray Inc., the Portland Group (PGI), and CAPS enterprise have announced a new parallel programming standard OpenACC that allows programmers to add directives to their code to indicate compiler what portions of the code need to run on GPU. Once available, this will ease the programming effort.

Recently researchers have shown successfully how such heterogeneous computing resources can be programmed for linear algebra applications [1]–[3]. There has been some effort in programming high performance computing devices for econometrics and financial applications [4]–[11]. In this paper, we focus on an estimation problem that is based on an indirect likelihood inference method [12]. This method relies on simulations from a model and on nonparametric density or regression function computations. The estimation method is useful in various econometrics and financial applications such as nonlinear models with latent variables (e.g., multivariate stochastic volatility models), where computational complexity can render methods such as maximum likelihood estimation infeasible. The method reduces computational complexity through the use of a statistic, which reduces dimensionality. It can still be very efficient, as long as the statistic is well chosen, so as to capture the information in the full sample. 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. [13], [14]). The paper of Creel and Kristensen makes clear the relationship with maximum likelihood estimation, and establishes a theoretical base for the estimators. 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.

We implemented the estimator on a machine with two 2.67GHz Intel Xeon X5650 processors and four NVIDIA M2090 GPU devices. A Tesla M2090 device has 512 CUDA Cores. We optimized the GPU code by using shared memory available to a block of threads. This reduces the memory traffic to the slow global memory of the GPU. The second major optimization we did was to improve effective utilization of the registers available to a thread by register blocking technique. We compared the optimized GPU code performance with a C based sequential version of the code that was executed on the host machine. We observed a speed up factor of up to 242 based sequential version of the code that was executed on the host machine. We observed a speed up factor of up to 242 based sequential version of the code that was executed on the host machine.

II. CUDA Programming Environment

A typical program on a system with a single GPU device is a C/C++ program with CUDA APIs to move data between system memory and GPU device memory, and to launch computation kernels on GPU. The data between system memory and the device memory is moved using the PCI Express (PCIe) bus. These transfers are costly and therefore applications that have a higher computation to I/O ratio are suitable for GPU computing. Also, if possible these transfers should be minimized and it is desirable to leave the data on GPU if a subsequent kernel is going to use the same data. A GPU device uses several memory spaces that differ in their size, access latency, and read/write restrictions. These memory spaces include global, local, shared, texture, and registers. Global, local, and texture memory have the greatest access latency, followed by constant memory, registers, and shared memory.

The GPU device works best for computations that can be executed concurrently on multiple data elements. In general, given an application one would like to partition the computation requirement into thousands of small computations that can be executed simultaneously. These computations are assigned to thousands of threads of GPU which are executed concurrently on different cores. When implementing applications on a system with multiple GPU devices, the approach for parallelization has to be adjusted. For this case, we partition the application in as many coarse-level chunks of computation as the number of devices available on the system. Next for each chunk, we partition the computation requirement as before into thousands of small computations that can be executed simultaneously.

CUDA provides an abstraction of thread hierarchy to allow computation from different domain to nicely map to different cores of the underlying hardware. The GPU hardware consists of a number of streaming multiprocessor which in turn consists of multiple cores. Threads are organized in blocks, where one or more block runs on a streaming multiprocessor. The

III. Estimator Based on Indirect Likelihood Inference

Indirect likelihood inference is a method of econometric estimation that relies on simulations from a model and on nonparametric density or regression function computations [12]. Following the notation of [12], we briefly describe the estimator. We wish to learn about a parameter \( \theta \) which is a function of the statistic. This is very much like a maximum likelihood estimator, except that the likelihood is normally not known on closed form. The maximum indirect likelihood estimator maximizes the indirect likelihood defined through \( Z_n \):

\[
\hat{\theta}_{MIL} = \arg \sup_{\theta \in \Theta} \log f_n(Z_n | \theta).
\] (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 \( k \), 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 [15], as it obviates the need for numerical optimization. One possibility is to use the posterior mean of \( \theta \) given \( Z_n \) defined as

\[
\hat{\theta}_{BIL} = \underset{\theta \in \Theta}{\mathbb{E}} f_n(Z_n | \theta) = \int_{\Theta} \theta f_n(Z_n | \theta) d\theta,
\] (2)

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

\[
f_n(Z_n | \theta) = \frac{f_n(Z_n | \theta) \pi(\theta)}{\int_{\Theta} f_n(Z_n | \theta) \pi(\theta) d\theta}
\] (3)

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:
A. SBIL

The BIL defined above (equation 2) is the posterior mean. The SBIL estimator proposed in Creel and Kristensen [12] (essentially the same idea was proposed in the ABC literature by [13]) directly computes the posterior mean using simulation and nonparametric regression, as follows. Make i.i.d. draws $\theta^s$, $s = 1, ..., S$, from the pseudo-prior density $\pi(\theta)$, for each draw generate a sample $Y_n(\theta^s)$ from a 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 $Z_S = \{Z_n^s, s = 1, 2, ..., S\}$ 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 neighbor regression estimator (see [16], Ch. 14).

$$\hat{\theta}_{SBIL} = \frac{1}{k} \sum_{s=1}^{S} \mathbf{1}(\| Z_n^s - Z_n \| \leq d_k(Z_n, Z_S)),$$

where $\mathbf{1}(-)$ 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.

IV. IMPLEMENTATION

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, $Z_n^j$, $j = 1, 2, ..., nZn$. $nZn$ 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 $Z_n^j$, $j = 1, 2, ..., nZn$, 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 $\theta^s$ that generated each of the neighbors. This is done using CUDA.

- **Step 4.** Once the neighbors are found and the $\theta^s$ 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 M2090 GPU device. In Matlab like notation, we will find it convenient to describe this computation as

$$[Z_s] = ma1_arp_stat(n, p, S)$$

Here, $Z_s$ is a $dimZ \times S$ matrix where $dimZ = p + 2$, and $S \leq 2^{10}$. Similarly, we represent the $k$ nearest neighbor computation as

$$[dist, ZZ] = knn(Z_S, Z_n, dimZ, k)$$

Here, $Z_n$ is a $nZn \times dimZ$ matrix, $dist$ is a $k \times nZn$ matrix, and $ZZ$ is a $dimTheta \times (nZn \times k)$ matrix where $nZn \leq 2^{10}$, and $dimTheta = p + 4$. Later we show how we generalize to solve large problems that do not fit in the device memory.

A. Computation of the simulated $Z_n^s$

Computation of $Z_n^s$ requires first sampling $\theta^s$ from the prior distribution $\pi(\theta)$, which involves generation of uniform random numbers. Then for a given $\theta^s$, 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 [17]. 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 $\theta^s$ 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. The OLS calculation is solving for $b$ in the following equation:

$$X'Xb = X'y,$$

where $b$ is a $(p+1) \times 1$, $X$ is a $n \times (p+1)$, and $y$ is $n \times 1$. Algorithm 1 outlines the OLS calculation.

Algorithm 1: OLS Calculation to estimate statistics for the AR$(p)$ model.

1. matmul($X$, $W$) // Compute matrix $X'X$ and store it in $W$
2. matvec($X$, $y$, $m$) // Compute the vector $m = X'v$
3. choleskyf($W$) // Compute upper triangular matrix $U$ such that $W = UU'$ using Cholesky method. Store $U$ and $U'$ in place
4. for $i = 0$ to $nZn$ do
5.   new = knn($Z$, $Zn$, $dimZ$, $k$)
6.   merge($dold$, $d$, $dnew$)
7.   merge($ZZold$, $ZZ$, $ZZnew$)
8.   $dold \leftarrow dnew$
9.   $ZZold \leftarrow ZZnew$
10. end

Algorithm 2: Algorithm to process large size problems.

1. $[Z_0] = ma1_arp_stat(S_0)$
2. for $j = 0$ to $nZn$ do
3.   $[dold_j, ZZold_j] = knn(Z_0, Zn_j, dimZ, k)$
4. end
5. for $i = 0$ to $S$ do
6.   $[Z_i] = ma1_arp_stat(S_i)$
7.   for $j = 0$ to $nZn$ do
8.     $[dj_j, ZZ_j] = knn(Z_i, Zn_j, dimZ, k)$
9.   end
10. end
11. end

B. Computation of $k$ nearest neighbors

For this computation we use an existing implementation available at http://www.i3s.unice.fr/~creative/KNN/ ( [18]). This code uses the brute force method of computing nearest neighbors, which involves explicitly computing the distances between all target and query points. There are sophisticated methods such as ANN that uses $kd$-tree data structure to reduce the number of distance computations [19]. However, in [20] authors demonstrated that sophisticated methods do not perform well on GPUs compared to the highly-parallelizable brute force method. We have modified the KNN code to use multiple GPU devices, if available.

C. Large Problem Size

To address large problem size, where $S > 2^{19}$ (or whatever limit the device supports) and $nZn > 1000$, 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.

A high-level Matlab like description of the algorithm is given in Algorithm 2. Essentially in the algorithm you iterate over $m$, where in each iteration you compute simulation statistics $Z_i$ for a $S_i$ followed by computation of $k$ nearest neighbors for the computed $Z_i$. We keep the results of $k$ nearest neighbors in a buffer that we update after each iteration.

D. Optimization

Use of shared memory. GPU device has shared memory which is hundred times faster than the global memory. Each streaming multiprocessor has its own shared memory, and all threads of a block running on a streaming multiprocessor share the same shared memory. The size of shared memory varies depending on the device and is typically in the range of 16 KB to 48 KB. Shared memory can also be viewed as a software programmable cache. In our implementation, a thread is responsible for generating one or more simulation statistics. The maximum benefit we observed for using the shared memory was using it in the OLS calculation. In OLS calculation we compute a matrix $W$ (Line 1 of Algorithm 1) which is used later in Line 3-5. Hence, $W$ is a good candidate for storing in shared memory. The threads in a block share the same memory and we need to partition the shared memory equally amongst all threads in a block. When allocating shared memory to a thread, we need to ensure that a thread only writes in its allocated space. It is advisable to allocate fixed space in the shared memory to a thread throughout its life cycle.

Register blocking using loop unrolling. Loop unrolling or blocking at registers level helps in improving the number of floating point operations to the number of load/stores [21]. To understand this, consider Line 1 of Algorithm 1 where we are
computing \( W = X'X \). A straightforward implementation of this function is shown in Algorithm 3, and a unrolled version of the code for \( p = 2 \) is shown in Algorithm 4. Execution of Line 6 of Algorithm 3 requires two loads and one store from the global memory; and two floating point operations resulting in a flops to load/stores ratio of 2/3. For the unrolled version of the code (Algorithm 4 for \( p = 2 \)), Lines 3 to 5 correspond to load data from global memory (or cache); and Lines 6 to 11 correspond to computations. Observe that unrolling for \( p = 2 \) improves the ratio of the number of floating point operations to the number of loads/stores from 2/3 to 12/3. Unrolling the code for \( p = 4 \) increases the ratio further to 6. In general, this ratio is given by \((p + 2)\). However, as we increase \( p \) we increase the required register count. Once we increase the registers requirement beyond the number of available registers for a thread (32 single precision registers), we start observing spilling in the compiled code that offsets the advantage of unrolling. Fortunately, for values of \( p \) of interest to us \((p < 11)\) we were able to take advantage of unrolling. Also, observe that in Algorithm 4 we exploit the symmetry of the matrices and are only performing about half of the floating point operations as compared to the Algorithm 3.

**Algorithm 3:** A straightforward implementation of \( \text{matmul}(X, W) \).

```
1 for i = 0 to p do
2   for j = 0 to p do
3      W[i * (p + 1) + j] = 0.0
4   end
5 for k = 0 to n - 1 do
6      W[i * (p + 1) + j] += X[k * (p + 1) + i] * X[k * (p + 1) + j]
7 end
8 end
```

**Algorithm 4:** An unrolled implementation of \( \text{matmul}(X, W) \) for \( p = 2 \).

```
1 w0=0.0, w1=0.0, w2=0.0, w3=0.0, w4=0.0 w5=0.0
2 for k = 0 to n - 1 do
3   x0 = X[k*n+n]
4   x1 = X[k*n+1]
5   x2 = X[k*n+2]
6   w0 += x0*x0
7   w1 += x0*x1
8   w2 += x0*x2
9   w3 += x1*x1
10  w4 += x1*x2
11  w5 += x2*x2
12 end
```

**V. Experimental Results**

We evaluated the performance for our implementation on a machine with two 2.67GHz Intel Xeon X5650 processors and four NVIDIA M2090 GPU devices. A Tesla M2090 device has 512 CUDA Cores. The GPU code was implemented using CUDA 4.0 programming environment. For comparison, we also wrote a C based sequential version of the code that was executed on the host machine. As outlined in Section 4, for parallel implementation we focused on the computation of simulated statistics (SS) and computation of nearest neighbors (NN) components of the overall computation. There are various parameters which determine the complexity of these two components. We find it convenient to identify the problem size for the two components by a 5-tuple \(<nZn, n, p, k, S>\).

Table 1 compare performance results for sequential implementation, initial implementation (non-optimized) on GPU with four devices, and an optimized implementation on GPU with four devices for problem size \(<3500, 400, 10, 400, 2^{20}>\) and \(<7000, 400, 10, 400, 2^{20}>\). For GPU implementation we have split the time for two components SS and NN. We have only optimized the SS component on GPU, and for NN we took the implementation as is from [18] with minor modification to run on multiple devices. Observe that the initial GPU implementation speeds up the sequential implementation by a factor of 93 for \(nZn = 7000\) and by a factor of 68 for \(nZn = 3500\). The optimized GPU implementation further improves the speed up factor to 242 for \(nZn = 7000\) and to 229 for \(nZn = 3500\). Note that all the improvement in the GPU implementation is due to the optimization of the SS component. The optimized GPU implementation speeds up the SS component by a factor of 7.5 compared to the non-optimized GPU implementation. Before optimization SS component is about 59% of the total time, and after optimization it is around 20% of the overall time. After optimization, the total time is dominated by the NN component, which is around 80% of the overall time. Thus for any further enhancement in the performance, we need to reduce NN component time.

We performed experiments to see the impact on the speed up with the number of GPU devices. Figure 2 illustrate speed up plots for three problem sizes: \(<7000, 400, 10, 400, 2^{18}>\), \(<7000, 400, 10, 400, 2^{19}>\), and \(<7000, 400, 10, 400, 2^{20}>\). We conducted experiments for very large simulated statistics up to \(2^{25}\). The results for this experiment are shown in Figure 3. Note that as we double the size of the simulated statistics, we almost double the total execution time.

**VI. Conclusion**

In this paper, we proposed an optimized implementation of an estimator based on indirect inference computation which has applications in the area of econometrics and finance. We demonstrated that it is possible to significantly speed up the computation using GPUs. We demonstrated that a system with four GPU devices can speed up the computation by a factor of 242 compared to a sequential non-optimized code. To enhance the performance further it is necessary to speed up
up the nearest neighbor (NN) component of the overall computation. The GPU kernels we have developed to implement the estimator have been written for two specific models, but are structured so as to be easily adapted to other models. Given the good speedups we have obtained for the two simple examples explored up to this point, we are optimistic that even better results will be obtained when more computationally demanding models are introduced. In the future, we plan to develop other related GPU kernels and make them available as a library for the econometrics and financial community to use.

REFERENCES

Fig. 3. Performance results for optimized implementation on GPU with four devices for problem sizes: $<7000, 400, 10, 400, S>$, where $S = 2^{L_S}$ and $L_S = 18$ to $25$.


