Optimal Bandwidth Selection for Kernel Regression Using a Fast Grid Search and a GPU

Chris Rohlfs
Morgan Stanley and Columbia University
car2228@columbia.edu

Mohamed Zahran
Computer Science Department
New York University
mzahran@cs.nyu.edu

***Abstract*—This study presents a new algorithm and corresponding statistical package for estimating optimal bandwidth for a non-parametric kernel regression. Kernel regression is widely used in Economics, Statistics, and other fields. The formula for the optimal "bandwidth," or smoothing parameter, is well-known. In practice, however, the computational demands of estimating the optimal bandwidth have historically been prohibitively high. Consequently, researchers typically select bandwidths for kernel regressions using *ad hoc* rules of thumb. This paper exploits the Single Program Multiple Data (SPMD) parallelism inherent in optimal bandwidth calculation to develop a method for computing optimal bandwidth on a GPU. Using randomly generated datasets of different sizes, this approach is shown to reduce the run time by as much as a factor of seven.**

Keywords*—*nonparametric, kernel, regression, optimal bandwidth, cross-validation, GPU.

I. Introduction

Nonparametric regression is used extensively in the field of econometrics to summarize relationships among variables with simple graphs. The approach is popular in part because it allows economists to relax restrictive assumptions about functional forms —*e.g.,* that the relationship between two variables is linear or log linear. In place of these assumptions, the researcher can simply assume that the relationship between the two variables is smooth (*i.e.*, continuously differentiable up to a certain order). One key factor that is relevant when performing nonparametric regression is to correctly choose the "bandwidth," or smoothing parameter. This smoothing parameter is selected to strike a balance between reducing the variance at each point (by increasing the number of nearby observations used to estimate the value) and reducing the bias toward smoothness (by focusing specifically on observations that are very close to the point at which we wish to estimate our conditional mean).

One factor limiting the popularity of nonparametric estimation, particularly for large datasets when it would be most valuable, is computational intensity. The computational intensity is especially pronounced for one important procedure: the selection of the optimal bandwidth (that minimizes the sum of the variance and squared bias). Due to this computational intensity, economists often use rule of thumb procedures in place of the optimal bandwidth (for examples in kernel density estimation, see [25], [26]). One notable feature about the computational requirements of optimal bandwidth selection is that they are well-suited for Single Program Multiple Data (SPMD) parallelism. The aim of this study is to exploit this potential for parallelism by designing code to perform optimal bandwidth selection using Graphics Processing Units (GPUs).

The standard approach for estimating the optimal bandwidth involves performing numerical optimization over different bandwidth values. Given that the objective function is not necessarily concave, however, this approach can produce unstable and unreliable results. It is consequently preferable to perform a "grid search," where we construct an evenly-spaced array of possible bandwidths (or an evenly-spaced grid or matrix in multivariate contexts), and the optimum is selected as the one from this array that produces the highest value of the expression being optimized. Supposing that the number of possible bandwidths is $O(n)$, however, and given that the objective function takes $O(n^2)$ steps to calculate, performing a standard grid search has complexity $O(n^3)$. We address this time cost with two strategies. First, we introduce a new sorting-based method for constructing the leave-one-out estimator for a grid of multiple bandwidths at once in $O(n \log n)$ time, reducing the complexity of the overall problem to $O(n^2 \log n)$. Second, we perform each of these $O(n \log n)$ steps in parallel on a GPU. As the number of cores in the machine approaches $O(n)$, all of these $O(n \log n)$ steps can be performed simultaneously, reducing the time cost further to the cost of the leave-one-out construction on a single core.

Section 2 of this paper presents a brief review of the literature on nonparametric econometrics and the application of parallelism and GPUs to economic and econometric problems. Section 3 provides a formal discussion of the operations to be performed and their complexity. Section 4 provides details on the implementation of this algorithm, including optimizations to enhance performance. Section 5 presents the results of this algorithm on randomly generated datasets of different sizes. The optimized CUDA program currently does not work for sample sizes greater than 20,000; however, at 20,000, it reduces the run time by a factor of seven relative to the method currently available in R for estimating optimal bandwidth. Section 6 concludes.

II. Literature Review

Nonparametric estimation is used in Economics and Statistics to provide visual summaries of population distribution

[0]Chris Rohlfs has a Ph.D. in Economics from University of Chicago in 2006, and M.Sc. in Computer Science from NYU in 2015. He is now Vice President of Wealth Management Risk at Morgan Stanley in New York City, and he is a part-time doctoral student in Deep Learning at Columbia University.

functions (pdfs) and statistical relationships without imposing restrictive assumptions about functional form *e.g.*, that a given distribution is normal or uniform or that the relationship between two variables x and y is linear or log linear. In place of these functional form assumptions, nonparametric estimation involves the assumption that a given pdf or functional relationship is sufficiently smooth that it is continuously differentiable up to a certain degree. In traditional parametric estimation, statistical results consist of specific parameter estimates —*e.g.*, standard deviations, coefficients —and standard errors, nonparametric results are typically presented graphically. Nonparametric density estimation or kernel density estimation (KDE) was first developed by Parzen [22] and Rosenblatt [24] and Nonparametric regression or kernel regression by Nadaraya [19] and Watson [27], all in the mid 20th century. The techniques were popularized in the 1990s and later as increased computational power and larger datasets made the techniques practical. In addition to being used extensively in Economics and Statistics, the methods are used in other areas such as Chemistry [8], Physics [9], and Machine Learning [1] and Computer Vision [30].

Both nonparametric density estimation and nonparametric regression involve the calculation of locally weighted statistics —either frequencies or conditional means —around a grid of points of interest, where the weighting kernel has a specific functional form such as a Gaussian distribution around each evaluation point in a grid. The degree of smoothing is determined by the bandwidth, which is selected to strike a balance between improving precision by using a larger number of observations and limiting smoothness bias by focusing on observations close to the evaluation points. Experts in the field prefer the leave-one-out cross-validation method for optimal bandwidth selection; however, practitioners typically use less computationally intensive rules of thumb. Li and Racine [18] and Pagan and Ullah [21] provide detailed explanations of the methodology, including references and applications.

One important feature about KDE and kernel regression that is relevant for the current study is that the techniques are highly amenable to parallelism, and in particular the Single Program Multiple Data (SPMD) parallelism that allows for computation on Graphics Processing Units (GPUs). GPUs have been used to parallelize a variety of computationally intensive procedures in Chemistry [3], Mathematics [29], Molecular Biology [14], and Physics [15], [16], including extensive applications to Markov Chain Monte Carlo (MCMC) estimation [6], [7], [14], [15], [28], a technique that is also used in Economics and Finance.

Despite the large amount of data and the many computationally intensive procedures used in applied economics and econometrics, however, there is relatively little work in that field that uses GPUs or high performance parallel computing. Some researchers have used the GPU for econometric applications including regression with missing data [2], estimating heteroskedasticity- and autocorrelation-corrected variance-covariance matrices [4], [5], and a Bayesian likelihood procedure applied to forecast terrorist activity in Colombia [28]. Other work has implemented multivariate regression with heteroskedasticity and autocorrelation-corrected standard errors on large datasets using a pipelining procedure on a field-programmable gate array (FPGA) [13], and one statistical study has implemented bootstrapping with a GPU [6]. In the broader fields of Economics and Finance, GPUs have been used in estimating general equilibrium macroeconomic models of the business cycle [11], options pricing [7], and projecting losses for insurance companies [20].

Of these studies, there is one that applies nonparametric regression on a GPU [10]. The authors apply these nonparametric estimates in the context of a new Bayesian procedure developed by the authors that they call indirect likelihood inference. While the code in that study is primarily written to implement the new statistical procedure, the authors provide a useful first pass at nonparametric estimation on the GPU, including some useful optimizations for efficient memory management. The authors use the k-nearest neighbor approach to nonparametric estimation —which is more amenable to SIMD parallelism —rather than the more common fixed-bandwidth kernel approach.

This study builds upon these authors contributions by developing a package for optimal bandwidth selection for kernel regression using a GPU. While the current implementation only uses one kernel weighting function, it is straightforward to add additional ones in the future. Additionally, the methods developed here for least-squares cross-validation can be applied to many similar problems in nonparametric estimation, including optimal bandwidth selection for kernel density estimation and the estimation of leave-one-out cross-validated confidence intervals for kernel density estimates and kernel regressions. To increase the accessibility of the contributions here to the applied econometrics and statistics communities, the `dynamic.load` package in R will be used to integrate the compiled C and CUDA programs for optimal bandwidth selection into an R package that will be made publicly available.

III. Conceptual Framework

Using observations of Y_i and X_i on *n* different observations, the researcher hopes to estimate the expected value of Y_i conditional on X_i for an array of different values of X_i using a bivariate kernel regression, using a nonparametric kernel weighting function *K* with smoothing parameter *h*. As Li and Racine [18] (pg. 69) discuss, the optimal bandwidth for a nonparametric kernel regression is determined by selecting the parameter *h* that minimizes the following equation:

$$CV_{lc}(h) = n^{-1}\sum_{i=1}^{n}(Y_i - \hat{g}_{-i}(X_i))^2 M(X_i), \qquad (1)$$

where $\hat{g}_{-i}(X_i)$ is defined to be the "leave-one-out estimator" of the mean of Y_i conditional on X_i, expressed as:

$$\hat{g}_{-i}(X_i) = \sum_{l\neq i}^{n} Y_l K((X_i - X_l)/h) / \sum_{l\neq i}^{n} K((X_i - X_l)/h) \quad (2)$$

and $M(X_i)$ is an indicator function for whether the denominator in (2) is non-zero. One of the most common kernel weighting functions is the Epanechnikov kernel [18] :

$$K(u) = 0.75 * (1 - u^2) * I\{|u| \leq 1\} \qquad (3)$$

where *I* denotes the indicator function

In this and most other standard kernels, the bandwidth *h* enters the equation in two ways: it scales the difference $X_i - X_l$, and it affects the indicator function in the kernel. If $X_i - X_l$ is greater than *h* for a given observation *l*, then that observation has no weight in the summation.

It is straightforward to see that doing so involves a tremendous amount of computation. In order to arrive at a $CV_{lc}(h)$ value for a single bandwidth *h*, it is necessary to perform a summation over all *n* in Equation (1), where each element in the summation is the ratio of two summations over all but one observation, as illustrated in Equation (2). This $O(n^2)$ procedure must be repeated for multiple bandwidth values.

Li and Racine [18] note that this minimization problem "can be solved using any standard numerical optimization procedure." It should also be noted, however, that if we do not assume a specific parametric distribution for the residuals $(Y_i - \hat{g}_{-i}(X_i))$, then the objective function in Equation (1) is not necessarily concave. Consequently, numerical optimization techniques such as Newton-Raphson will often produce non-global minima that depend upon the initial values used in the calculations. Hence, a reliable procedure for estimating the optimal bandwidth may require a more computationally intensive approach such as a grid search, making our optimization problem $O(k*n^2)$, where *k* is the number of points in the grid. Supposing that *k* is proportional to the number of observations *n*, we have an $O(n^3)$ problem.

This study introduces two new strategies for reducing the complexity of this problem. First, we note that much of the computation for the leave-one-out estimator is independent of the value of *h* and need not be repeated across iterations. Consider the case of a single leave-one-out estimator $(Y_i - \hat{g}_{-i}(X_i))$, using the Epanechnikov kernel from Equation (3). For a given value of *h*, it is necessary to sum each of Y_l, $Y_l * (X_i - X_l)^2$, and $(X_i - X_l)^2$, across all but one of the observations that satisfy $(X_i - X_l) \leq h$. The second and third summations must then be divided by h^2. Next, note that, for every $h_2 > h_1$, every term that appears in the summations for h_1 also appears in the summations for h_2. Hence, if the data are sorted in order of $(X_i - X_l)^2$, then once the summations are complete for the first bandwidth value h_1, we use the same summations for bandwidth h_2 and add the terms for the remaining observations that satisfy $(X_i - X_l) \leq h_2$. This procedure can be continued for all of the bandwidth values being considered. Using this strategy, it is possible to consider a grid of $O(n)$ bandwidths and to compute all $O(n^2)$ leave-one-out estimators in $O(n \ \log \ n)$ time, where the summation requires $O(n)$ operations and the $O(n \ \log \ n)$ sorting algorithm dominates the computation. With this optimization alone, the problem is simplified from $O(n^3)$ to $O(n^2 \ \log \ n)$.

The second strategy that we introduce involves the use of parallelism. Given the SPMD nature of the problem, we can construct $(Y_i - \hat{g}_{-i}(X_i))$ for each of the different *i* values in parallel using a many core machine like a GPU. Each core can compute the summations described above for all of the possible bandwidths in $O(n \ \log \ n)$ time. As the number of cores in the machine approaches $O(n)$, then the time cost of computing the optimal bandwidth falls further from its original cost of $O(n^3)$ to $O(n \ \log \ n)$.

IV. Experimental Setup

We suppose that the researcher provides data on Y_i and X_i and will use an Epanechnikov kernel.[1] The Nadaraya-Watson [19], [27] local constant estimator is used rather than a local linear regression. The Nadaraya-Watson estimator is the most commonly used kernel regression estimator and is the default in the common R package `np` [23]. An evenly spaced grid of possible bandwidth values is considered. The user may specify the minimum and maximum values in the grid as well as the number of bandwidths to consider. As a default, the maximum bandwidth in the grid is the domain of X_i (*i.e.,* the difference between the maximum and minimum values), and the minimum bandwidth is that domain divided by the number of bandwidths being considered.

The program for estimating optimal bandwidth is written in C, using the CUDA API. While the functions may accommodate any pair of Y_i and X_i vectors, we use randomly generated data to test the performance of the different algorithms. X_i is assumed to be uniformly distributed between zero and one, and Y_i is defined to equal $0.5 * X_i + 10 * X_i^2 + u_i$, where u_i is uniformly distributed between zero and 0.5.

A. Memory Allocation

The optimization procedure developed in this program requires a considerable amount of the GPU's global memory. First, and less importantly from a resource standpoint, we copy the arrays of size *n* of Y_i and X_i values and the array of size *k* of possible bandwidths into memory, where *k* is the number of bandwidths being considered. Another array of size *k* holds the final cross-validation scores for each bandwidth, which is used to determine which bandwidth is optimal.

In addition to these relatively minor demands on memory, we generate several matrices in global memory that are used for intermediate calculations. It may be possible to eliminate the need for some of these intermediate matrices as the algorithm is optimized, but appear in the program as it currently stands. We allocate memory for two *n* by *n* matrices, one of $abs(X_i - X_l)$ values and another of Y_i values. The purpose of these matrices is to allow each thread to generate its own observation-specific sorted versions of these arrays in order to construct the summations used to construct the leave-one-out estimator. these matrices are used to construct the *n*-by-*k* sums over *i* of $Y_l * (X_i - X_l)^2/h^2$, and $(X_i - X_l)^2/h^2$, which are key components in the construction of the observation-specific leave-one-out estimators. These terms are combined to produce another *n*-by-*k* matrix, this one of the squared differences $(Y_i - \hat{g}_{-i}(X_i))^2$ that appear as elements of the summation in Equation (1), each one appropriately adjusted by the indicator function $M(X_i)$.

Allocating memory for these many matrices —especially the *n* by *n* ones —not only involves a large time cost, it severely

[1]The sorting strategy described in the previous section is not restricted to the Epanechnikov kernel. The same approach can be used for the Uniform and Triangular kernels. The Gaussian, which is probably the second most common kernel weighting function, does not use an indicator function to exclude observations and can consequently be constructed for *k* different bandwidths without the need for a sort.

limits the size *n* of datasets for which the program can be applied. Later versions of this study will examine ways to reduce the use of these intermediate data objects and also to make use of more recent compute capability GPUs that allow dynamic allocation of global memory within the device kernel; the current version was built to ensure compatibility across different GPUs and drivers. Additionally, later versions of this study will consider alternative ways to make more efficient use of the GPU memory, swapping matrices out to the host memory or to disk as necessary. To reduce the demands for global memory and to ensure compatibility with relatively early GPUs and NVCC drivers, only single-precision floating point numbers are used in the computation.

Once these various matrices are allocated in the GPU memory, the Y_i and X_i arrays are copied into the device's global memory. The array of bandwidth values is the same for all observations and is consequently stored in the GPU's constant memory. Because the typical GPU's cache working set for constant memory is only 8 KB, no more than 2,048 bandwidth values can be considered in the optimization. If a higher level of precision is necessary, the user can run the optimization code multiple times with progressively smaller ranges of possible bandwidths.

B. Sequence of Operations

The main kernel constructs estimates of the leave-one-out estimator separately for each observation. Each thread *j* fills in *n* values of the $abs(X_i - X_j)$ and Y_i matrices. Next, it sorts both of these matrices in order of $abs(X_i - X_j)$. Each thread performs its own complete sort. An iterative variant of QuickSort is used, modified from [12] to sort floating point numbers and to also sort an auxiliary variable. This iterative QuickSort improves upon the recursive version by eliminating the need for a tree of recursive subcalls with the associated additions to the memory stack. Additionally, using the iterative version helps to maintain compatibility with earlier GPUs, as earlier versions of CUDA do not allow functions to contain recursive sub-calls.

After these arrays are sorted, they are used to populate the *n*-by-*k* summation matrices. We begin with the smallest bandwidth. We begin with the first observations in the array, with the smallest values of $abs(X_i - X_j)$. For each observation *i* with $abs(X_i - X_j)$ less than that smallest bandwidth, we add $(X_i - X_j)^2$ to the corresponding in our first *n*-by-*k* matrix corresponding to observation *j* and the smallest bandwidth, and we add Y_i times that value to corresponding element in our second *n*-by-*k* matrix. Once we reach an observation whose value of $abs(X_i - X_j)$ exceeds the smallest bandwidth, we move onto the next bandwidth value. We copy our sums for the smallest bandwidth value over to the elements corresponding to observation *j* and the second smallest bandwidth, and we begin to add additional values to that sum, starting with the next observation *i*, provided that its value of $abs(X_i - X_j)$ does not exceed the second smallest bandwidth. This procedure continues until each thread *j* has computed each of its bandwidth-specific sums.

Next, each thread *j* loops through all *k* bandwidths. For each one, both sums *j*-specific sums are divided by the square of the bandwidths and are multiplied by 0.75. Following the design of the leave-one-out estimator, the final sums exclude the values corresponding to the j^{th} observation. These sums are then combined with the data on Y_j to construct squared residuals of the form $(Y_j - \hat{g}_{-j}(X_j))^2 M(X_j)$ separately for each bandwidth. To facilitate efficient caching of memory and to reduce bank conflicts, the matrix indices are switched at this stage. When the summations are first constructed, they are used in loops over the different bandwidths. For this reason, they appear in groups of *k*, with a separate group for each observation *j*. The squared residuals are to be summed across *j* separately for each bandwidth, however. Hence, the array is indexed as *k* separate groups of *n*.

Because this main kernel does not use shared memory or coordination across threads, the block size and grid size were selected to minimize the run-time. The total number of threads in the grid was set equal to the number of observations in the data. The fastest performance was found with threads per block set to 512, the maximum possible on the GPU being used, so that number of threads per block was chosen.

Once the squared residuals have been calculated, our remaining tasks consist of reductions. We first add up the thread-specific squared residuals separately by bandwidth, and we then determine which bandwidth has the lowest sum of squared residuals. These reductions are performed using a modified version of the C and CUDA code that appears in [17]. That code employs a variety of strategies, including extensive use of shared memory and unrolling of loops in a radix sum to minimize the time cost of the reduction. First, a summation reduction is performed *k* times, once for each bandwidth, to add up the squared residuals and consequently to arrive at bandwidth-specific cross-validation scores $CV_{lc}(h)$ for different values of *h*. Supposing that the number of threads per block is *T*, a single block is called, and *T* elements are stored in shared memory. Each thread *t* first adds together the values of $(Y_j - \hat{g}_{-j}(X_j))^2 M(X_j)$ for the observations *j* for which *j* equals *t* modulus *T*. Then, the threads synchronize, and each thread with $t < T/2$ adds to its sum the sum from the thread $t + T/2$. The process repeats with $T/4,\ T/8$, and so on until thread zero contains the full sum.

Next, a variation of the reduction code is used that determines the minimum among those *k* different scores. In order to identify the bandwidth corresponding to the minimum cross-validation score, it is necessary to store $2{*}T$ elements in shared memory. The first *T* contain the cross-validation scores, and the next *T* contain the bandwidths to which they correspond.[2] The sequence of operations is similar. A single block is called, and each thread *t* computes the minimum value of $CV_{lc}(h)$ among those bandwidths whose index values (rank when sorted) equal *t* modulus *T*. Each time one of these minima is updated with a new value of $CV_{lc}(h)$, the shared memory at position $t + T$ is updated with the value of the bandwidth corresponding to that cross-validation score. After performing the reduction

[2]The same operation can be performed without copying the bandwidths into shared memory, if we simply save the integer-value of the thread index with the minimum bandwidth value —and then we can simply access that element of the bandwidth array in the main memory after the procedure. In fact, it should be possible to avoid saving the bandwidth values entirely. All we need, in order to reconstruct the possible bandwidths, are the minimum value, the maximum value, and the number of bandwidths to be considered. Given that the program is so memory-intensive, reconstructing these bandwidth values with each iteration should have little effect on the overall runtime.

minimum across the various threads in the block, element *T* in the array of shared memory contains the optimal bandwidth value.

C. Testing Design

The correctness of the program was ensured in multiple ways. First, the sequential C code and the CUDA code were checked against each other to ensure that they produced identical results under many different sets of inputs. A variety of bugs were identified through these comparisons. The debugging process also involved examining many intermediate steps in the code to ensure that, when considering sample sizes for which hand calculation was feasible, the various sums and minima were actually the intended sums and minima. Finally, while the R programs used different randomly generated data, an additional check on the correctness of the algorithms was to verify that both R programs produced optimal bandwidths in similar ranges to what was obtained from the C and CUDA code.

To evaluate the run time of our GPU-parallelized program for optimal bandwidth selection, we consider four programs:

1) **Racine & Hayfield.** The optimal bandwidth selector provided in the R package `np` [23]. Note that one of the authors of this package is also an author of [18]).
2) **Multicore R.** An optimized cross-validation bandwidth selector constructed by the author in the R programming language. This program makes use of the `data.table` and `parallel` libraries to speed up some of the operations and to perform some of the operations concurrently.
3) **Sequential C.** A sequential version of the C code used here to estimate the optimal bandwidth.
4) **CUDA on GPU.** The parallelized version of the C code using CUDA to estimate the optimal bandwidth on the GPU.

Both 1) and 2) are programmed in R and identify the optimal bandwidth using built-in numerical optimization techniques in R. For this reason, both programs may produce estimates that are not global minima and are sensitive to the initializing values. Indeed, in [23] the authors suggest that it may be useful to run the algorithm multiple times with different initial values to ensure that one obtains a global solution. Program 1) is the benchmark program and is publicly available. Programs 2), 3), and 4) were created by the author of the current study. Program 3) is similar to 4), the main program being evaluated. It uses the iterative QuickSort approach described here to perform a grid search across possible bandwidth values.

All four programs are tested on a machine with 16 2.53 GHz Intel Xeon CPU cores, 16 GB of main memory, and two Tesla S10 GPUs, each with 240 streaming cores and 4 GB of device-specific GPU memory.

Sample sizes of n = 100, 500, 1,000, 5,000, 10,000, and 20,000 are considered. Beyond that point, the GPU could not allocate the memory required for the intermediate matrices. Future versions of this paper will attempt to address this limitation by reducing the reliance on global memory allocations. For programs 3) and 4), numbers k of different bandwidths are considered, including k = 5, 10, 50, 100, 500, 1,000, and 2,000, with the number of bandwidths never exceeding the number of observations used. As discussed earlier, the number of bandwidths may not exceed 2,048 due to the 8 KB upper bound on the cache working set for constant memory in most GPUs. The range of possible bandwidths used is the default for the programs, with the largest possible bandwidth equaling one, because the domain of the uniformly distributed X_i is [0,1]. The smallest possible bandwidth is one divided by the number of bandwidths being estimated.

For each program, sample size, and number of bandwidths combination, the program is run five times around a similar time as the other programs are run to ensure that the system loads and other factors influencing run times are similar for the different programs. Run times for the R programs are computed inside the interacting R environment on Linux using the `system.time` command. Hence, these run times do not include the time cost of generating the data vectors. For the C and CUDA programs, run times are estimated through the Linux command `time`, applied to the compiled executable files and consequently do include the time costs of generating the random data. These $O(n)$ operations account for a small portion of the run-times and should have relatively little effect on the results.

V. Results

Figure 1 presents the main findings from this analysis. The four curves shown on the graph illustrate the run times of the four programs listed in the previous section as they vary with the number of observations n, plotted along the horizontal axis in logarithmic scale. The run time for each (program, n) pair is plotted in seconds along the vertical axis. The graph shows substantial speedups when moving from programs 1) to 2), 2) to 3), and 3) to 4), especially at the larger values of n. When $n = 20,000$, the highest value for which the optimal bandwidth is computed here, the runtime for the program 1), the existing R package, is 232.5 seconds, or almost four minutes. Program 2), the multicore R program introduced here, appears to be less efficient in its computations but makes up for that inefficiency with its use of 16 cores, giving it a run time of 124.7 seconds, or 44% lower than that of the benchmark. When using the sorting-based approach with the grid search over 50 possible bandwidths, when $n = 20,000$, the sequential code completes in 80.9 seconds, or 65% faster than the benchmark, and the code run on the GPU completes in 32.5 seconds, slightly less than one seventh of the time of the benchmark program. The speedup increases dramatically with the sample size n, but program 4) cannot run at sample sizes greater than 20,000, because the memory requirements become prohibitive. Future work will address this issue by eliminating the reliance on storing n-by-n matrices in the GPU's device memory.

The same run times are illustrated in Table I. As in the graph, the C programs each compute cross-validation scores for 50 different bandwidths. As the table shows, the C programs with the sorting-based approach, in addition to producing guaranteed global minima, are consistently faster than the R programs that use numerical optimization methods. At lower sample sizes, the sequential programs are faster than their parallelized counterparts. For both the R and the C programs, the run times for the sequential and parallelized programs are roughly equal around $n = 1,000$, and for n values greater than 1,000, the parallelized code is considerably faster.

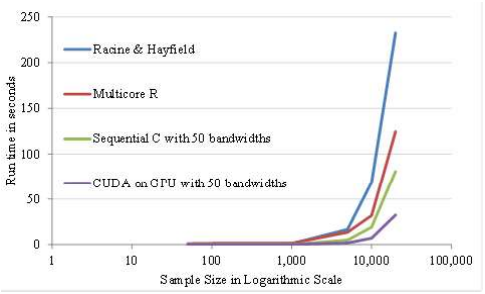

Fig. 1. Run Times by Program and Sample Size

TABLE I. RUN TIMES BY PROGRAM AND SAMPLE SIZE

Sample Size	Racine & Hayfield	Multicore R	Sequential C	CUDA on GPU
50	0.04	1.16	0.00	0.09
100	0.05	1.43	0.01	0.09
500	0.38	1.46	0.07	0.15
1,000	1.12	1.49	0.27	0.24
2,000	16.71	13.59	4.89	1.83
10,000	68.69	32.08	19.24	7.10
20,000	232.51	124.70	80.92	32.49

The next set of results, presented in Table II, illustrates how the run times of Programs 3) and 4) vary with the number of bandwidths for which the cross-validation scores are calculated. Results are shown for the sequential program in panel A and for the parallelized CUDA program in panel B. In both cases, rows correspond to numbers of bandwidths, and columns correspond to sample sizes, so that the number of bandwidths being considered increases as one moves down a column. For the sequential program in panel A, at relatively small sample sizes, the number of bandwidths has an appreciable effect on performance. For instance, when $n = 1,000$, the run time increases by roughly 70%, from 0.24 to 0.41, as the number of bandwidths increases from 5 to 2,000. The effect is relatively minor at large sample sizes; when $n = 20,000$, as the number of bandwidths increases from 5 to 2,000, the run time increases by less than 5%. For the CUDA program, we do not observe appreciable slowdowns associated with increasing the numbers of bandwidths for any sample size.

TABLE II. RUN TIMES BY NUMBER OF BANDWIDTHS CALCULATED

Panel A: Sequential C Program

	Sample Size						
Bandwidths	50	100	500	1,000	5,000	10,000	20,000
5	0.00	0.00	0.06	0.24	4.83	19.09	80.24
10	0.02	0.01	0.06	0.27	4.93	19.43	80.43
50	0.04	0.01	0.07	0.27	4.89	19.24	80.92
100		0.01	0.07	0.28	4.86	19.26	80.77
500			0.10	0.34	5.04	19.81	81.80
1,000				0.41	5.32	20.06	82.48
2,000					5.66	21.05	84.11

Panel B: CUDA Program Run on GPU

	Sample Size						
Bandwidths	50	100	500	1,000	5,000	10,000	20,000
5	0.09	0.09	0.15	0.24	1.80	6.94	31.83
10	0.09	0.09	0.15	0.24	1.82	7.00	32.08
50	0.09	0.09	0.15	0.24	1.83	7.10	32.49
100		0.09	0.15	0.25	1.84	7.11	32.56
500			0.16	0.26	1.86	7.13	32.55
1,000				0.26	1.92	7.32	33.13
2,000					2.05	7.68	34.21

VI. CONCLUSION

Optimal bandwidth estimation is an important input into the estimation of nonparametric regressions, and researchers often omit this key step, due to the computational complexity of the problem. This paper introduces two new innovations to reduce the run time required to estimate optimal bandwidths. First, a new sorting approach is introduced that makes a grid search possible with little increase in the run time, thus eliminating the need for relatively unreliable numerical optimization methods. Second, a program is introduced to compute an optimal bandwidth for an Epanechnikov kernel in parallel using a GPU. Relative to the publicly available R package, these optimizations together reduce the run time of this algorithm by a factor of 7.

As the results show, much of the reduction in run time was obtained by switching to C and by using the sorting-based grid search approach rather than the numerical optimization. That is, Programs 1), 2), and 3) each perform an $O(n^2)$ algorithm, but Program 3), which introduced those innovations, completed in much less time than Program 1) or than the sequential version of Program 2). Nevertheless, much of the code really is amenable to SPMD parallelism, and using CUDA really did speed up the estimation.

REFERENCES

[1] A. Amrouche, M. Debyeche, A. Taleb-Ahmed, J. Rouvaen, and M. Yagoub. An efficient speech recognition in adverse conditions using nonparametric regression. *Engineering Applications of Artificial Intelligence 23*(1), Feb. 2010.

[2] G. Beliakov, M. Johnstone, and S. Nahavandi. Computation of high breakdown regression estimators without sorting on graphics processing units. *Computing 94*(5), May 2012.

[3] R. Betz and R. Walker. Implementing continuous integration software in an established computational chemistry software package. In *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, May 2013.

[4] X. Cai and X. Lin. Forecasting high dimensional volatility using conditional restricted Boltzmann machine GPU. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, May 2012.

[5] X. Cai, G. Lai, and X. Lin. Forecasting large scale conditional volatility and covariance using neural network on GPU. *J Supercomputing 63*(2), Feb. 2013.

[6] D. Cheng and Y. Liu. Parallel Gibbs sampling for hierarchical Dirichlet processes via gamma processes equivalence. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug. 2014.

[7] G. Chow, A. Tse, Q. Jin, W. Luk, P. Leong, and D. Thomas. A mixed precision Monte Carlo methodology for reconfigurable accelerator systems. In *Proceedings for the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2012.

[8] P. Constans and J. Hirst. Nonparametric regression applied to quantitative structure-activity relationships. *J Chem Inf Comput Sci 40*(2), Mar. 2000.

[9] K. Cranmer. Kernel estimation in high-energy Physics. *Comput Phys Commun 136*(3), Mar. 2001.

[10] M. Creel and M. Zubair. High performance implementation of an econometrics and financial application on GPUs. In *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis*, Nov. 2012.

[11] M. Dziubinski and S. Grassi. Heterogeneous computing in Economics: A simplified approach. *Computational Economics 43*(4), Apr. 2014.

[12] D. Finley. Optimized quicksort —C implementation (non-recursive), 2010. Available at: http://alienryderflex.com/quicksort/

[13] C. Guo and W. Luk. Pipelined HAC estimation engines for multivariate time series. *J Sign Process Syst 77*(1-2), Oct. 2014.

[14] E. Hailat, K. Rushaidat, L. Schwiebert, K. Mick, and J. Potoff. GPU-based Monte Carlo simulation for the Gibbs ensemble. In *Proceedings of the High Performance Computing Symposium*, 2013.

[15] C. Hall, W. Ji, and E. Blaisten-Barojas. The Metropolis Monte Carlo method with CUDA enabled graphics processing units. *J Computational Physics 258*, Feb. 2014.

[16] A. Harju, T. Siro, F. Canova, S. Hakala, and T. Rantaliho. Computational Physics on graphics processing units. In *Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012*, Jun. 2012.

[17] M. Harris. Optimizing parallel reduction in CUDA. NVIDIA, Sep. 2012. Available at: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[18] Q. Li and J. Racine, *Nonparametric Econometrics: Theory and Practice*, 1st ed. Princeton, NJ: Princeton University Press, 2007.

[19] É. Nadaraya. On nonparametric estimates of density functions and regression curves. *Theory of Probability & Its Applications 10*(1), 1965.

[20] B. Norkin. Systems simulation analysis and optimization of insurance business. *Cybernetics and Systems Analysis 50*(2), Mar. 2014.

[21] A. Pagan and A. Ullah, *Nonparametric Econometrics*, 1st ed. Cambridge, UK: Cambridge University Press, 1999.

[22] E. Parzen. On estimation of a probability density function and mode. *Annals of Mathematical Statistics 33*(3), 1962.

[23] J. Racine and T. Hayfield. Package 'np.' Nonparametric kernel smoothing methods for mixed data types, Jul. 2014. Available at: http://cran.r-project.org/web/packages/np/np.pdf

[24] M. Rosenblatt. Remarks on some nonparametric estimates of a density function. *Annals of Mathematical Statistics 27*(3), Sep. 1956.

[25] S.J. Sheather and M.C. Jones. A reliable data-based bandwidth selection nmethod for kernel density estimation. *Journal of the Royal Statistical Society, Series B (Methodological) 53*(3), 1991.

[26] B.W. Silverman. *Density estimation for statistics and data analysis*. New York: Chapman and Hall, 1991.

[27] G. Watson. Smooth regression analysis. *Sankhyā 26*(15), Dec. 1964.

[28] G. White and M. Porter. GPU accelerated MCMC for modeling terrorist activity. *Computational Statistics and Data Analysis 71*, Mar. 2014.

[29] R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, eds. *Parallel Processing and Applied Mathematics: 9th International Conference*, Sep. 2011.

[30] S. Zhu, L. Zhang, and H. Jin. A locally linear regression model for boundary preserving regularization in stereo matching. In *Computer Vision —ECCV 2012: 12th European Conference on Computer Vision*, Oct. 2012.