Robust regression with CUDA and its application to plasma reflectometry

Diogo R. Ferreira,[1] Pedro J. Carvalho,[2] Horácio Fernandes,[2] and JET contributors[a)]
EUROfusion Consortium, JET, Culham Science Centre, Abingdon, OX14 3DB, UK
[1)]*Instituto Superior Técnico (IST), Universidade de Lisboa, Campus do Taguspark, Avenida Prof. Dr. Cavaco Silva, 2744-016 Porto Salvo, Portugal*
[2)]*Instituto de Plasmas e Fusão Nuclear (IPFN), Instituto Superior Técnico (IST), Universidade de Lisboa, Av. Rovisco Pais, 1049-001 Lisboa, Portugal*

(Dated: 20 November 2015)

In many applications, especially those involving scientific instrumentation data with a large experimental error, it is often necessary to carry out linear regression in the presence of severe outliers which may adversely affect the results. Robust regression methods do exist, but they are much more computationally intensive, making it difficult to apply them in real-time scenarios. In this work, we resort to GPU-based computing to carry out robust regression in a time-sensitive application. We illustrate the results and the performance gains obtained by parallelizing one of the most common robust regression methods, namely Least Median of Squares (LMedS). Although the method has a complexity of $O(n^3 \log n)$, with GPU computing it is possible to accelerate it to the point that it becomes usable within the required time frame. In our experiments, the input data comes from a plasma diagnostic system installed at JET (Joint European Torus, the largest fusion experiment in Europe), but the approach can be easily transfered to other applications.

I. INTRODUCTION

Microwave reflectometry is a plasma diagnostics technique which can be used to determine the density profile of a magnetically confined plasma inside a fusion device.[1–3] In essence, reflectometry consists in probing the plasma with an EM wave of frequency f_i, which is reflected at some cutoff position r_i inside the plasma where there is a density $n_e(r_i)$. While the density $n_e(r_i)$ that causes total reflection can usually be predicted from f_i, the actual cutoff position r_i where such reflection occurs is unknown and must be determined based on the group delay of the reflected wave.

The density profile (i.e. the density values $n_e(r_i)$ for a series of positions r_i with $i = 0, 1, \ldots, n-1$) can be obtained by sweeping the plasma with a series of frequencies $f_0, f_1, ..., f_{n-1}$ in the microwave range. In the reflectometry system installed at JET,[4] this series of frequencies is divided into four bands, known as Q (41.6–53.6 GHz), V (50.4–76.8 GHz), W (72.0–114.6 GHz) and D (108.0–136.8 GHz). In each band, the probing frequency is increased from f_i to f_{i+1} in fixed steps of 0.01 to 0.02 GHz, and for each f_i the system records the amplitude and phase of the reflected wave, in the form of an I/Q signal.[5] Of special interest is the *beat frequency* at which such signal appears to oscillate, which is essentially proportional to the group delay.[6] The beat frequency is usually obtained through spectrogram analysis of the I/Q signal, specifically using the short-time Fourier transform (STFT).[7]

A critical issue in plasma reflectometry is the need to determine how much of the group delay is due to propagation of the probing signal inside the plasma region, and how much is due to dispersion in the waveguides that carry the probing signal to the plasma region. Eventually, the delay component that is due to dispersion must be subtracted from the measured group delay in order to obtain the propagation time of the probing signal within the plasma region.

The delay due to dispersion is usually determined through a calibration procedure that takes place just before the presence of plasma, and which consists in sweeping the empty chamber with the same set of probing frequencies.[8,9] When the chamber is empty, the probing signal is reflected on the back wall,[10] and since the back wall is at a fixed position, any variation in group delay must be due to dispersion. This makes it possible to determine the behavior of dispersion delay across the entire range of probing frequencies, and to correctly compensate for its effect at each probing frequency.

Figure 1 shows the results obtained in the Q-band for a sample sweep of the empty chamber. The x-axis is the probing frequency, and the y-axis is the beat frequency of the signal reflected on the back wall. Since the beat frequency is obtained by Fourier analysis, the results are displayed in the interval $[-\frac{f_s}{2}, \frac{f_s}{2}[$ where $f_s = 200$ MHz is the sampling frequency.

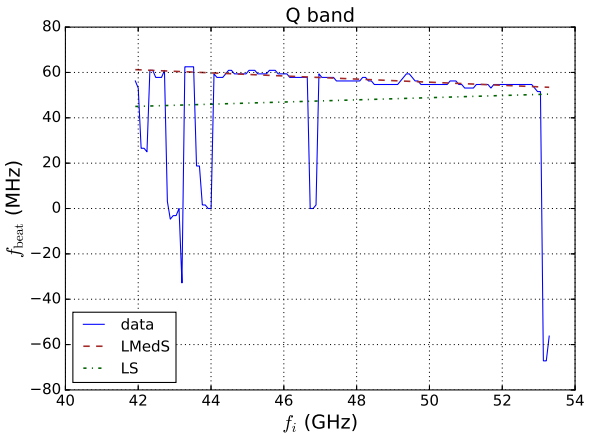

FIG. 1. Beat frequencies for a sweep of the empty chamber (Q-band)

As is apparent in Figure 1, there is an overall linear trend between probing frequency and beat frequency from the back wall. It is this linear trend that must be captured for the pur-

[a)]See the Appendix of F. Romanelli et al., Proceedings of the 25th IAEA Fusion Energy Conference 2014, Saint Petersburg, Russia

pose of system calibration. However, Figure 1 also shows that an ordinary least squares regression (LS) provides a poor fit to the data. This is due to significant deviations (outliers) that arise from experimental error or even to the presence of some particles in the supposedly empty chamber.

A much better fit to the data can be obtained using a robust regression method such as least median of squares (LMedS).[11] This method will be discussed in Section II. However, robust regression methods are much more computationally intensive because they must fit a subset of the data (i.e. a subset that excludes the outliers). Finding this subset is a computationally demanding task, which becomes a major obstacle when trying to perform system calibration in real-time.

For example, if the calibration sweep is performed at $t = 39$ seconds, and the plasma starts entering the chamber at $t = 40$ seconds, then there is less than 1 second to calculate the beat frequencies and to find the fitting line. As we will see in this paper, just finding the fitting line takes more than 1 second even on a fast CPU. Therefore, we turn to GPU (Graphics Processing Unit) computing, and in particular to NVIDIA's Compute Unified Device Architecture (CUDA)[12] to develop a parallel implementation that is able to carry out the whole data processing in much less than 1 second, as described in Section III. At the end of Section III, we also discuss the use of multi-threading and multiple GPUs to achieve the same goal, before concluding the paper in Section IV.

II. ROBUST REGRESSION

Ordinary least squares (LS) regression consists in finding the line that minimizes the sum of the squared residuals over all data points, i.e.:

$$\min_{a,b} \sum_{i=1}^{n} [y_i - (ax_i + b)]^2 \tag{1}$$

where a and b are the line parameters (slope and y-intercept, respectively) and the data points are represented as (x_i, y_i) with $i = 1, 2, ..., n$. The problem with this method is that a single outlier can drive the fitting line away from all other points if its residual is allowed to be arbitrarily large.

There are two main approaches to deal with this problem:

- The first approach is to consider only the h smallest residuals in the sum of Eq. (1). This method is known as Least Trimmed Squares (LTS).[11] Its advantage is that the $n-h$ largest residuals (which are possibly due to outliers) are left out of the sum, and therefore have no influence on the fitting line. The disadvantage is that this method is equivalent to finding the subset of h elements which yields the best fit. Since there are $\binom{n}{h}$ subsets to choose from, in general it is impractical to carry out an exhaustive search. Some algorithms provide an approximate solution by resorting to sub-sampling or random search.[13,14] Exact solutions of LTS can be computed in $O(n^3 \log n)$, and with some refinements this can be brought down to $O(n^2 \log n)$.[15] Another issue with LTS is the choice of h, which can be adjusted in the interval $\frac{n}{2} < h \leq n$ to determine the breakdown point of the method (i.e. the proportion of outliers that the method will be able to tolerate).

- The second approach to robust regression is to replace the sum of squared residues in Eq. (1) with the median of those squared residues. This method is known as Least Median of Squares (LMedS).[11] Its robustness comes from the fact that the actual residues that are either above or below the median do not matter; only the midpoint residue is used to decide between several candidate solutions. An interesting result about LMedS is that the slope of the fitting line is equal to the slope of a line that passes through a pair of data points.[16] This suggests that it is possible to find the fitting line by searching through all pairs of data points (there are $n(n-1)/2$ such pairs). The complexity of this approach is $O(n^3 \log n)$,[16] but there are some algorithms for LMedS with $O(n^2 \log n)$ or even less.[17] LMedS has the largest possible breakdown point (it tolerates up to 50% of outliers in the dataset).

For our purposes, we need not the fastest algorithm, but one which can be easily parallelized to take advantage of the parallel processing capabilities of the GPU. Instead of trying to converge to a solution through some iterative method, which would be inherently sequential, from the perspective of parallel computing it is better to explore a search space where each candidate solution can be evaluated independently from (and therefore in parallel with) every other.

From the two methods described above, LMedS has some practical advantages over LTS. In terms of search space, LTS involves $\binom{n}{h}$ subsets of points, whereas LMedS involves only $\binom{n}{2}$ pairs of points; also, the median is lighter to compute than the sum of trimmed squares; and, finally, LMedS provides a high robustness without requiring a choice of h, as in LTS.

For these reasons, we adopted LMedS in our work, and we use the following simple algorithm which, in practice, yields good results and can be parallelized with relative ease:

- for each pair of points (x_i, y_i) and (x_j, y_j) with $j > i$,
  - calculate the parameters (a, b) of the line $y = ax + b$ that passes through (x_i, y_i) and (x_j, y_j),
  - compute the squared residue $[y_k - (ax_k + b)]^2$ for every point (x_k, y_k) in the dataset,
  - sort the residues and find the median residue,
- pick the line which yields the least median residue.

From this description it becomes clear why the complexity of this method is $O(n^3 \log n)$. There are $n(n-1)/2$ pairs of points, so that is $O(n^2)$. For each pair, we need to compute all residues, so that is a further $O(n)$. However, the next step (sorting the residues) is more computationally intensive since, on average, it takes $O(n \log n)$ to sort an array. Therefore, the overall complexity is $O(n^2) \times O(n \log n) = O(n^3 \log n)$.

III. IMPLEMENTATION WITH CUDA

CUDA[12] is a technology introduced by NVIDIA® to make the parallel computing capabilities of GPUs accessible for general-purpose programming. In general, the GPU has a lower clock speed than the CPU, but modern GPUs have hundreds or even thousands of cores, so they can largely outperform the CPU in certain parallelizable tasks.

CUDA is both an architecture and a programming model. The architecture refers to the design and internal structure of the GPU hardware, while the programming model refers to how software can be developed on top of that hardware.

Internally, GPU cores are organized into processing units called *streaming multiprocessors*.[18] Each streaming multiprocessor has a multiple of 32 cores (the exact multiple depends on the CUDA architecture version), and a GPU can have as many as a dozen multiprocessors or even more (the exact number depends on the GPU model). For example, the mid-range GeForce GTX 750 Ti has 5 streaming multiprocessors with 128 cores each, for a total of 640 cores.

In terms of programming model, CUDA revolves around the idea of dividing work into a large number of independent threads. The actual work to be carried out by each thread is programmed into a special function called a *kernel*. A kernel is basically a C function that executes on the GPU and is replicated into as many threads as necessary. Each thread has a unique *thread id* which can be used to distinguish between the multiple executions of the same kernel.

Typically, a CUDA kernel reads data from one or more input arrays, and writes data to one or more output arrays. All of these arrays must be allocated on GPU memory. The thread id is used to determine exactly on which elements of the input and output arrays each thread will operate. Thus, by launching a sufficiently large number of threads, an entire dataset can be processed in parallel according to the instructions of a given kernel. The processing can also be done in several stages, where each stage consists in invoking a different kernel, and the output of one kernel becomes an input to the next, as we will see below.

A. LMedS with CUDA

The LMedS algorithm described in the previous section can be parallelized by first noting that each pair of points can be processed independently of every other. For a given pair of points (x_i, y_i) and (x_j, y_j) we can use a thread to calculate the parameters of the line $y = ax + b$ that passes through those two points. Furthermore, each single residue can be computed independently of every other; for a given point (x_k, y_k) and for given line parameters (a, b) we can use a thread to compute the squared residue $[y_k - (ax_k + b)]^2$. Finally, to sort the residues and to find the least median residue there are also techniques that can employed to take advantage of the parallel capabilities of the GPU.

The first step is to calculate the line parameters and this is implemented as illustrated in Figure 2. There are two input arrays of size n, where n is the number of points. One of the arrays contains the x-coordinates and the other contains the y-coordinates of the data points. Each thread picks up a pair of points (x_i, y_i) and (x_j, y_j) and calculates the line parameters (a, b). Since there are $n(n-1)/2$ pairs, the number of threads and the size of the output array is also $n(n-1)/2$.

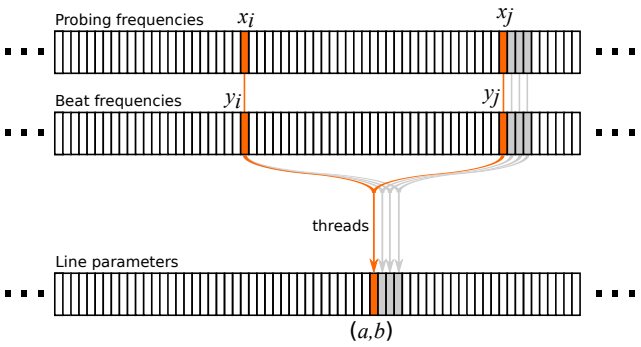

FIG. 2. Calculating the line parameters (kernel #1)

The second step is to compute the residues, and we do this not only for a single line, but for every point and every line at once. This is implemented in a second kernel, and is illustrated in Figure 3. Here, there are three input arrays, namely the x- and y-coordinates that also served as input to the first kernel, and the line parameters that were produced as output from the first kernel. Since there are n points and $n(n-1)/2$ lines, the total number of residues to be computed is $n^2(n-1)/2$. Therefore, the number of threads and the size of the output array is also $n^2(n-1)/2$.

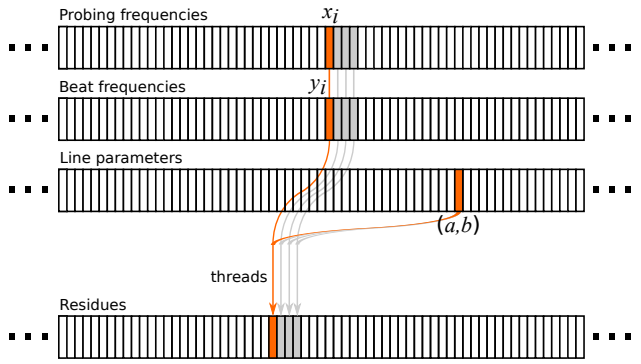

FIG. 3. Computing the residues (kernel #2)

The third step is to sort the residues for each line. For this purpose we note that, in the previous kernel, the residues for any given line are stored contiguously in a segment of the output array. Therefore, the sorting must be applied within each of those segments. Specifically, there are $n(n-1)/2$ segments of size n to be sorted. Although there are fast algorithms to sort a single large array on the GPU,[19] here we have many short segments instead. If possible, this should be done all at once on the GPU, rather than sorting the segments one-by-one which, even on the GPU, would be rather slow.

For this purpose we use the Modern GPU library,[20] which provides a segmented sort algorithm that runs efficiently on the GPU. The algorithm is a variant of the merge sort that does

not swap elements across the boundaries between segments. To use this algorithm, we need to define where the boundaries are (basically, there is a new boundary after every n elements), and then the algorithm sorts the entire array at once on the GPU, as illustrated in Figure 4. This sorting occurs in-place, so the input and output arrays are the same.

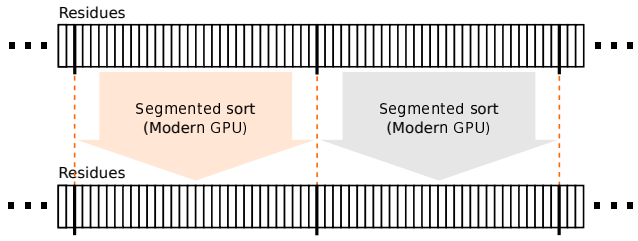

FIG. 4. Segmented sort (with the Modern GPU library)

The fourth step is to collect the median residue for each line. This is a matter of writing a kernel to retrieve the middle element from each sorted segment, as depicted in Figure 5. The input array is of size $n^2(n-1)/2$, and the output array is of size $n(n-1)/2$.

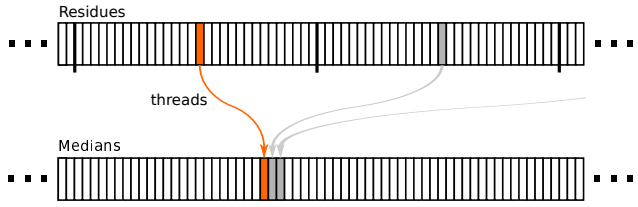

FIG. 5. Median residue for each line (kernel #3)

In a fifth and final step we need to find the least median residue (i.e. the minimum) in the array of median residues. The actual value of such minimum is irrelevant; only its position matters. Since the segmented sort algorithm that we applied before keeps the order of segments, we can go back to the array of line parameters to find, at exactly the same position, the values of (a, b) that correspond to the least median residue. This is illustrated in Figure 6. To find the position of the minimum element in an array in GPU memory, we use the `min_element()` function available in the Thrust library.[21]

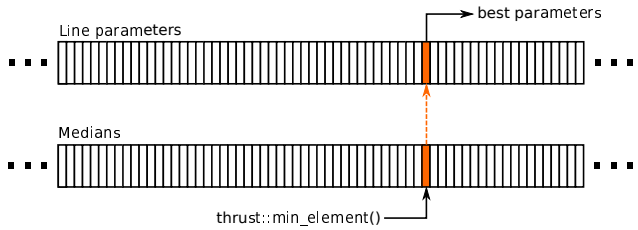

FIG. 6. Least median residue (with the Thrust library)

B. Performance gain

To assess the performance of our CUDA implementation, we compare it with a CPU-based version written in C. Basically, this C version is a direct implementation of the LMedS algorithm described in Section II; it uses `for` loops to go through each pair of points and to compute the residues, and it uses the quick sort algorithm (`qsort` function) from the C standard library to sort the residues.

For benchmarking purposes, we run both versions – the CUDA version and the C version – on two different machines. The first machine is a relatively recent desktop PC with an i5 processor and a mid-range GPU; the second machine is a server PC with an i7 processor and a somewhat older GPU, which was top-of-the-line some years ago (2010).

Table I lists the hardware details and the results obtained on the first machine. There are four frequency bands to be processed separately, meaning that LMedS is applied four times to find the fitting line from the data points in each of those bands. Table I indicates the number of points and the run time for both the CUDA version and the C version.

CPU: Intel Core i5-4690 @ 3.5–3.9 GHz GPU: NVIDIA GeForce GTX 750 Ti with 640 CUDA cores @ 1137 MHz				
Frequency band (GHz)	No. points (n)	C version t_{CPU} (ms)	CUDA version t_{GPU} (ms)	Performance gain t_{CPU}/t_{GPU}
Q (41.6–53.6)	143	61.420	3.730	16.5×
V (50.4–76.8)	243	327.417	16.419	19.9×
W (72.0–114.6)	268	441.181	22.276	19.8×
D (108.0–136.8)	268	435.372	22.078	19.7×
Overall		1265.395	64.509	19.6×

TABLE I. Benchmarks on the desktop PC

Except for the Q-band, where the performance gain is slightly lower, the CUDA version is about 20× faster. The total time required to process the four bands is about 1.27 seconds on the CPU, but only 64 ms on the GPU.

Table II lists the results obtained on the second machine. Here, the CPU appears to be slightly slower, but the GPU is faster. These two effects combine to achieve a performance gain that, in most cases, is above 40×. In terms of total processing time, the CPU takes about 1.45 seconds to process the four bands, whereas the GPU takes a mere 34 ms.

CPU: Intel Core i7-3820 CPU @ 3.6–3.8 GHz GPU: NVIDIA GeForce GTX 480 with 480 CUDA cores @ 1401 MHz				
Frequency band (GHz)	No. points (n)	C version t_{CPU} (ms)	CUDA version t_{GPU} (ms)	Performance gain t_{CPU}/t_{GPU}
Q (41.6–53.6)	143	70.278	2.527	27.8×
V (50.4–76.8)	243	374.848	8.598	43.6×
W (72.0–114.6)	268	502.891	11.532	43.6×
D (108.0–136.8)	268	499.023	11.449	43.6×
Overall		1447.047	34.120	42.4×

TABLE II. Benchmarks on the server PC

C. Multi-threading and multiple GPUs

In the previous benchmarks, we compared our CUDA implementation with a single-threaded C version that did not take advantage of the fact that it was running on multi-core processors. (There are four physical cores in both the i5 and the i7, with the i7 exposing eight logical cores through hyper-threading.) A natural question is how much performance can be achieved by making use of multiple cores.

In particular, there are four bands to be processed, and each band can be handled by a separate thread. On a quad-core processor such as the i5, it should be possible to use all of its cores to process the four bands in parallel. For this purpose, we developed a multi-threaded C version where the LMedS routine is inside a thread function that is instantiated four times with a call to `pthread_create()` from the POSIX threads library. Although the distribution of work across CPU cores is not controlled by the program, it is expected that the CPU will make use of several cores to run those threads.

Table III shows the results of running the multi-threaded C version on both machines. The performance gain is relative to the single-threaded C version on the same machine, whose run times have been reported earlier in Tables I and II. Here, the processing time for each band is more or less the same. However, the total processing time is shorter, since it is roughly equal to the run time of the longest thread. This yields a performance gain of about 2.8×, which is far below of what it is possible to achieve with the CUDA version.

	Desktop PC (i5-4690)		Server PC (i7-3820)	
Band	Multi-threaded C version $t_{\text{CPU*}}$ (ms)	Performance gain $t_{\text{CPU}}/t_{\text{CPU*}}$	Multi-threaded C version $t_{\text{CPU*}}$ (ms)	Performance gain $t_{\text{CPU}}/t_{\text{CPU*}}$
Q	65.059	0.94×	70.317	1.00×
V	339.223	0.97×	377.414	0.99×
W	450.393	0.98×	509.589	0.99×
D	445.554	0.98×	493.349	1.01×
Overall	450.435	2.81×	509.654	2.84×

TABLE III. Benchmarks with the multi-threaded C version

In an analogous reasoning to the use of multiple cores, one could argue that further performance gains could be achieved in the CUDA version by making use of multiple GPUs. For this purpose, we installed a second GPU on both machines, and developed a multi-GPU CUDA version that is also multi-threaded in the sense that there are two CPU threads dispatching work to the two GPUs. Since there are four bands, each GPU is assigned two bands to process.

Table IV shows the results obtained with the multi-GPU version on both machines. Again, the performance gain is relative to the single-threaded C version on the same machine. Comparing the run times in Table IV to those of the CUDA version in Tables I and II, one observes that the processing time and performance gain for each band are roughly the same. However, with two GPUs now working in parallel, the total processing time is lower, yielding an improvement of about 1.5× over the single-GPU version.

	Desktop PC (2× GTX 750 Ti)		Server PC (2× GTX 480)	
Band	Multi-GPU CUDA version $t_{\text{GPU*}}$ (ms)	Performance gain $t_{\text{CPU}}/t_{\text{GPU*}}$	Multi-GPU CUDA version $t_{\text{GPU*}}$ (ms)	Performance gain $t_{\text{CPU}}/t_{\text{GPU*}}$
Q	3.877	15.8×	2.493	28.2×
V	16.639	19.7×	9.086	41.3×
W	22.323	19.8×	11.557	43.5×
D	22.265	19.6×	11.706	42.6×
Overall	40.978	30.9×	23.145	62.5×

TABLE IV. Benchmarks with the multi-GPU CUDA version

From these results, one can extrapolate that a multi-GPU CUDA version running on four GPUs would yield a maximum improvement of 3× over the single-GPU version, which is comparable to what was obtained when going from the single-threaded C version to the multi-threaded one.

IV. CONCLUSION

GPU computing can bring dramatic performance improvements to real-time diagnostics. Even for algorithms that are polynomial in time, as is the case with LMedS regression, the large number cores in a GPU can bring the execution time down to a fraction of what it would be on a CPU. The more data- or processing-intensive an algorithm is, the more benefit can potentially be drawn from a GPU-based implementation. However, for this purpose the problem must be cast into a form that allows each data element to be processed independently, and therefore in parallel with every other.

Our implementation of LMedS with CUDA has five main stages, where at each stage a different kernel function or library routine performs parallel processing of one or more input arrays into an output array. The hardware that we used consists of commonly available graphics cards, so it should be possible to obtain even higher performance gains with more powerful and specialized GPUs. In any case, this illustrates the immense potential of these technologies to perform computationally-intensive tasks under stringent time requirements. The use of robust regression as a calibration procedure in plasma reflectometry is an example of such scenario.

In future work, we will be looking into the possibility of parallelizing other tasks in plasma reflectometry, namely the calculation of the density profile,[22] with a view towards enabling the use of such technique as a real-time diagnostic in magnetic fusion devices.

ACKNOWLEDGMENTS

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. IST activities also received financial support from *Fundação para a Ciência e a Tecnologia* through project UID/FIS/50010/2013.

[1]F. Simonet, Review of Scientific Instruments **56**, 664 (1985).
[2]C. Laviron, A. J. H. Donné, M. E. Manso, and J. Sanchez, Plasma physics and controlled fusion **38**, 905 (1996).
[3]L. Meneses, L. Cupido, A. Sirinelli, M. E. Manso, and JET-EFDS Contributors, Review of Scientific Instruments **79** (2008).
[4]A. Sirinelli, B. Alper, C. Bottereau, F. Clairet, L. Cupido, J. Fessey, C. Hogben, L. Meneses, G. Sandford, M. J. Walsh, and JET-EFDA Contributors, Review of Scientific Instruments **81** (2010).
[5]S. Hacquin, L. Meneses, L. Cupido, N. Cruz, L. Kokonchev, R. Prentice, and C. Gowers, Review of Scientific Instruments **75**, 3834 (2004).
[6]A. Silva, M. E. Manso, L. Cupido, M. Albrecht, F. Serra, P. Varela, J. Santos, S. Vergamota, F. Eusébio, J. Fernandes, T. Grossmann, A. Kallenbach, B. Kurzan, C. Loureiro, L. Meneses, I. Nunes, F. Silva, W. Suttrop, and the ASDEX Upgrade Team, Review of Scientific Instruments **67**, 4138 (1996).
[7]W. L. Zhong, Z. B. Shi, X. L. Zou, X. T. Ding, X. L. Huang, Y. B. Dong, Z. T. Liu, W. W. Xiao, X. Q. Ji, Z. Y. Cui, Y. Liu, L. W. Yan, Q. W. Yang, and X. R. Duan, Review of Scientific Instruments **82** (2011).
[8]T. Estrada, J. Sánchez, B. Van Milligen, L. Cupido, A. Silva, M. E. Manso, and V. Zhuravlev, Plasma physics and controlled fusion **43**, 1535 (2001).
[9]X. Weiwen, L. Zetian, D. Xuantong, S. Zhongbing, and V. Zhuravlev, Plasma Science and Technology **8**, 133 (2006).
[10]G. F. Matthews, M. Beurskens, S. Brezinsek, M. Groth, E. Joffrin, A. Loving, M. Kear, M.-L. Mayoral, R. Neu, P. Prior, V. Riccardo, F. Rimini, M. Rubel, G. Sips, E. Villedieu, P. de Vries, M. L. Watkins, and EFDA-JET contributors, Physica Scripta **2011**, 014001 (2011).
[11]P. J. Rousseeuw, Journal of the American Statistical Association **79**, 871 (1984).
[12]J. Nickolls, I. Buck, M. Garland, and K. Skadron, ACM Queue **6**, 40 (2008).
[13]P. J. Rousseeuw and M. Hubert, Lecture Notes-Monograph Series , 201 (1997).
[14]E.-W. Bai, Automatica **39**, 1651 (2003).
[15]O. Hössjer, Computational Statistics & Data Analysis **19**, 265 (1995).
[16]J. M. Steele and W. L. Steiger, Discrete Applied Mathematics **14**, 93 (1986).
[17]D. L. Souvaine and J. M. Steele, Journal of the American Statistical Association **82**, 794 (1987).
[18]J. Nickolls and W. J. Dally, IEEE micro **30**, 56 (2010).
[19]H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, in *Parallel Processing and Applied Mathematics*, LNCS, Vol. 6067 (Springer, 2010) pp. 403–410.
[20]S. Baxter, "Modern GPU," (2013), http://nvlabs.github.io/moderngpu/.
[21]NVIDIA Corporation, "Thrust quick start guide," (2014), http://docs.nvidia.com/cuda/thrust/.
[22]E. Mazzucato, Review of Scientific Instruments **69**, 2201 (1998).