GPU acceleration for statistical gene classification

GPU Acceleration for Statistical Gene Classification

Alfredo Benso, Stefano Di Carlo, Gianfranco Politano, and Alessandro Savino
Politecnico di Torino, I-10129, Torino, Italy
Department of Control and Computer Engineering
email: {alfredo.benso,stefano.dicarlo,gianfranco.politano, alessandro.savino}@polito.it

***Abstract*—The use of Bioinformatic tools in routine clinical diagnostics is still facing a number of issues. The more complex and advanced bioinformatic tools become, the more performance is required by the computing platforms. Unfortunately, the cost of parallel computing platforms is usually prohibitive for both public and small private medical practices. This paper presents a successful experience in using the parallel processing capabilities of Graphical Processing Units (GPU) to speed up bioinformatic tasks such as statistical classification of gene expression profiles. The results show that using open source CUDA programming libraries allows to obtain a significant increase in performances and therefore to shorten the gap between advanced bioinformatic tools and real medical practice.**

***Index Terms*—GPU acceleration, gene expression, statistical classification, clinical diagnostics**

I. Introduction

Over the past few years, the amount of biological information generated by the scientific community has explosively grown thanks to important advances in both molecular biology and genomic technologies. To be fruitfully analyzed and exploited, this vast amount of data requires both advanced specialized tools and powerful computing platforms. Bioinformatic is the computer science discipline that strives to solve these problems. A common characteristic of Bioinformatic algorithms is therefore the high data dimensionality and the repetitive execution, either sequential or recursive, of heavy computational cycles. One example is DNA microarray analysis, one of the fastest-growing technologies in the field of genetic research. DNA microarrays are small solid supports, e.g., membranes or glass slides, on which sequences of DNA are fixed in an orderly arrangement. Tens of thousands of DNA probes can be attached to a single slide and used to analyze and measure the activity of genes. Scientists are using DNA microarrays to investigate several phenomena from cancer to pest control. DNA microarrays allow to measure changes in gene expression and thereby learn how cells respond to a disease or to a particular treatment [1], [2]. However, due to the large amount of information on their surface, microarrays' analysis is a complex task. It is in fact very expensive, in terms of computational costs, to thoroughly analyze all variables and their correlations. For example, in statistical classification of gene expression data, most classifiers need features reduction in order to reduce the complexity of the task and make it manageable by the available computational platforms [3], [4].

This paper proposes to exploit off-the-shelf Graphic Processor Units (GPUs) to accelerate DNA Microarray classification. The main benefit of having accelerated classification algorithms stems in the possibility of avoiding features reduction, thus minimizing the probability of discarding information potentially useful for the analysis. A Graphics Processing Unit is a processor commonly available in modern graphics adapters, and designed to efficiently compute floating point operations. Today, parallel GPUs have begun making computational inroads against the CPU, and a subfield of research, dubbed GPGPU for General Purpose Computing on GPU, has found its way into diverse fields [5], [6], [7]. Parallel distribution of tasks among several dedicated cores heavily improves time performances of an algorithm.

In this paper, a parallel classification algorithm has been developed on top of Gene Expression Graphs (GEG) presented in [8]. The proposed GEG based classifier, already presented in [8] in its non-parallel version, works by comparing GEGs actually represented by adjacency matrices. This perfectly matches two critical requirements of a "GPU enabled" algorithm: first, dealing with large floating point matrices, and second parallelizing repetitive arithmetical operations among identically structured data.

The paper is organized as follows: section II describes the background of the target algorithm; Section III presents the parallel computational platform and the parallelized algorithm. Results are presented in Section IV, while Section V concludes the paper suggesting future possible developments of this work.

II. GEG Based Classifier

A. Gene Data Modeling

A microarray experiment typically assesses a large number of DNA probes (e.g., genes, cDNA clones, or Expressed Sequence Tags - ESTs) providing gene expression levels under multiple conditions. The multivariate response of a microarray can be utilized as an electronic fingerprint to characterize a wide range of phenomena, and to build prediction algorithms for classifying diseases based on gene expression information. This process involves several steps including Signal preprocessing of raw microarray scans, Data modeling, Prediction (e.g., classification), and Validation [1], [9].

In [10] and [8] we proposed a new graph-based data model for groups of gene expression profiles built from raw gene expression measures. The model is constructed in order to allow efficient classification, and to avoid the influence of preprocessing steps on the prediction process.

In particular, a set of microarray experiments Tr can be modeled by a non-oriented weighted graph (Gene Expression

Graph) $GEG_i = (V_i, E_i)$ where:

- each vertex $v_x \in V_i$ represents a gene. Only vertices representing *relevant* genes are included in the graph;
- each edge $(u, v) \in E_i \subseteq V_i \times V_i$ connects pairs of vertices representing genes that are co-relevant, i.e., concurrently relevant, within a single sample. It therefore models relationships among relevant genes of a sample. If n genes are co-relevant in the same sample, each corresponding vertex will be connected with an edge to the remaining $n-1$ ones, thus creating a clique;
- the weight $w_{u,v}$ of each edge $(u, v) \in E_i$ corresponds to the number of times genes u and v are co-relevant in the same sample over the set of samples. In a graph built over a single experiment, each edge will be weighted as 1. Adding additional microarrays will modify the graph by introducing additional edges and/or by modifying the weight of existing ones.

The identification of relevant genes is usually performed by comparing diseased and healthy tissues. Complementary DNA (cDNA) microarrays [11], for example, provide for each gene two expressions using two different fluorescence intensities: one labeled Cy5 producing a red fluorescence and associated with a diseased condition, and one labeled Cy3 producing a green fluorescence and associated with a healthy condition. The predominance of one of the colors indicates the abundance of the corresponding DNA sequence, allowing us to introduce the concept of over-expressed or silenced genes. On the other hand, equal intensities indicate no peculiar information in the corresponding gene expression. The (binary) logarithm of the ratio $Cy5/Cy3$ (log-ratio) is usually used to express differential-expression of genes. A positive log-ratio identifies over-expressed genes while a negative log-ratio indicates a silenced gene.

A *Cumulative Relevance Count* (CRC) can be computed for each node v (gene) of the graph to reflect its expression trend across the experiments as follows:

$$CRC_v = \sum_{\forall sample \in Tr} Rel_v\,(sample) \qquad (1)$$

where $Rel_v(sample)$ is +1 if v is over-expressed in $sample$, -1 if v is silenced in $sample$, and 0 if v is not relevant in $sample$. A gene is considered relevant iff its CRC is not zero.

Fig. 1 shows an example of GEG construction from a set Tr of six samples (Fig. 1-a). Each sample includes 4 genes, and for each gene the Cy5 and Cy3 expression components are provided. Fig. 1-b shows the log-ratio calculated for each gene in each sample, and the indication of over-expressed relevant genes, silenced relevant genes, and non-relevant genes. Fig. 1-c shows the corresponding GEG, where each vertex corresponds to a gene that is relevant in at least one experiment. To give an example of how to compute the CRC for each vertex, and the weight of each arc, let us look in more details at vertices C and D. Looking at the log-ratio table, one can see that gene C is over-expressed in 3 experiments (Exp. 1, 3, and 5),

(a) Initial training set expression levels represented as a gene expression matrix

$$GEM = \begin{array}{c|cccc} Exp/Gene & A(Cy5,Cy3) & B(Cy5,Cy3) & C(Cy5,Cy3) & D(Cy5,Cy3) \\ Exp.1 & 5000,10 & 20,11100 & 15000,80 & 90,13000 \\ Exp.2 & 8000,20 & 20,12000 & 1000,1050 & 100,12000 \\ Exp.3 & 10000,10099 & 30,30000 & 11000,30 & 40,1900 \\ Exp.4 & 1200,20 & 15,10 & 10,100 & 8000,50 \\ Exp.5 & 5000,100 & 20,4500 & 10500,30 & 12500,70 \\ Exp.6 & 7000,15 & 70,5500 & 10100,10050 & 40,12500 \end{array}$$

(b) Log-ratios matrix and relevance with ε=1, and μ=1.

Exp/Gene	A	B	C	D
Exp.1	8.97	−0.12	7.55	−7.17
Exp.2	8.64	−9.23	−0.07	−6.91
Exp.3	−0.02	−9.97	8.52	−5.57
Exp.4	5.90	0.58	−3.32	7.32
Exp.5	5.64	−7.82	8.45	7.48
Exp.6	8.87	−6.30	0.01	−8.29

over-expressed
silenced

(c) Gene expression graphs.Silenced nodes indicate nodes with negative CRC, i.e., nodes silenced in the majority of the samples, while over expressed node represent nodes with positive CRC, i.e., nodes over-expressed in the majority of the samples.

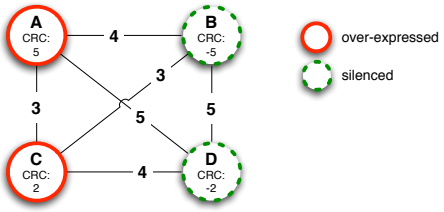

Figure 1. GEG construction example starting from the initial set of gene expression profiles, to the final graph construction

silenced in one experiment (Exp. 4), and non-relevant in two experiments (Exp. 2, and 6). The CRC of node C in the GEG is therefore $CRC_C = 3-1 = 2$. Gene D, instead, is silenced in 4 experiments, and over-expressed in 2 experiments: its CRC is therefore -2. To compute the weight of the edge (C, D) it is enough to count the number of experiments in which both genes are relevant (this time without taking into account the sign). They are experiments 1, 3, 4, and 5; the weight $w_{C,D}$ is therefore 4.

If new samples become available from new experiments referring to the same pathology, the related information can be easily added to the corresponding GEG without any additional memory requirement; GEGs memory occupation is in fact determined by the number of considered genes, only, and is independent of the number of experiments in the data-set.

B. Classification

Gene Expression Graphs represent an excellent data structure for building efficient classifiers. The classifier works by structurally comparing pairs of GEGs: one representing a given pathology (GEG_{pat}), built from a corresponding training set Tr_{pat} of known samples, and one representing the sample $\vec{s}$ to classify (GEG_s). This comparison measures how much GEG_s is similar (or can be overlapped) to GEG_{pat} in terms of over-expressed/silenced genes (CRC of vertices), and relationships between gene expressions (weight of edges). The result of this operation is a *proximity score* ($Ps \in [-1, 1] \subset \mathbb{R}$), computed according to eq. 2, measuring the similarity between the two graphs.

$$Ps(GEG_{pat}, GEG_s) = \frac{SMS(GEG_{pat}, GEG_s)}{MMS(GEG_{pat})} \quad (2)$$

SMS (sample matching score) analyzes the similarity of GEG_{pat} and GEG_s considering those vertices (genes) appearing in both graphs, only.

SMS is computed as:

$$SMS\,(GEG_{pat}, GEG_s) = \\ = \sum_{\forall (i,j) \in E_s \cap E_{pat}} \left[\left(Z_i \cdot w_{i,j} \cdot \frac{|Z_i|}{|Z_i| + |Z_j|} \right) + \right. \\ \left. + \left(Z_j \cdot w_{i,j} \cdot \frac{|Z_j|}{|Z_i| + |Z_j|} \right) \right] \quad (3)$$

where (i, j) are edges appearing in both GEG_s and GEG_{pat}, while Z_x is the z-Score of vertex v_x computed as:

$$Z_x = CRC_{x_{pat}} \cdot CRC_{x_s} \quad (4)$$

By construction, each vertex v_x of a GEG has $CRC_x < 0$ if g_x is silenced in the majority of the samples of its training set, $CRC_x = 0$ if g_x is actually not relevant in its training set, or $CRC_x > 0$ if g_x is over-expressed in the majority of the samples of its training set. The z-Score may therefore assume the following values:

- $Z_x > 0$: if g_x is silenced/over-expressed in both GEG_s and GEG_{pat};
- $Z_x < 0$: if g_x is silenced in GEG_s and over-expressed in GEG_{pat}, or viceversa;
- $Z_x = 0$: if g_x is not relevant either in GEG_s, or in GEG_{pat}.

MMS (maximum matching score) is the maximum SMS that would be obtained with all genes in GEG_s perfectly matching all genes in GEG_{pat}, with the z-Score of each gene always positive.

$$MMS\,(GEG_{pat}) = \\ = \sum_{\forall (i,j) \epsilon E_{pat}} \left(w_{i,j} \cdot \frac{CRC_i^2 + CRC_j^2}{|CRC_i| + |CRC_j|} \right) \quad (5)$$

C. Classifier software implementation

Since the GEG classifier has to compare each pathology (represented by a GEG_{pat}) with each sample to classify, the memory required for each comparison has to be enough to store two GEGs, each of several thousands of vertices (genes). In order to reduce the required amount of memory GEGs are never explicitly represented as a matrix or a list. Instead, we use a matrix where each line stores the genes' relevance of a single experiment. Therefore the actual software representation of a GEG is a matrix of experiments and their gene relevance values.

Algorithm 1 shows the MMS computation algorithm. For each row of the GEG matrix (line 4) the MMS contribution of each gene is computed (line 7) w.r.t. the set of the remaining genes (line 6).

Algorithm 1 MMS algorithm

```
1:  MMS (n_rows, n_genes, GEGMatrix, CRCVect)
2:  mms = 0
3:  for r=0 to n_rows-1 do
4:    for i=0 to n_genes-1 do
5:      for j=i+1 to n_genes - 1 do
6:        mms += MMScontribution(GEGMatrix[r][i],
          GEGMatrix[r][j])
7:      end for
8:    end for
9:  end for
10: return mms
```

Algorithm 2 SMS algorithm

```
1:  SMS (n_rows, n_genes, GEGMatrix, CRCVect, SAM-
    PLEVect)
2:  sms = 0
3:  for r=0 to n_rows-1 do
4:    for i=0 to n_genes-1 do
5:      for j=i+1 to n_genes - 1 do
6:        sms += SMScontribution(GEGMatrix[r][i], GEG-
          Matrix[r][j], SAMPLEVect[i], SAMPLEVect[j])
7:      end for
8:    end for
9:  end for
10: return sms
```

Resorting to the same approach, the SMS evaluation algorithm is defined by Algorithm 2. The GEG matrix exploration remains the same, where the sample vector (SAMPLEVect in the algorithm) is used to compute each SMS contribution, according to eq. 3.

III. GPU Acceleration

A. CUDA

Nvidia's Compute Unified Device Architecture (CUDA) [12] is a software platform for massively parallel high-performance computing on the company's powerful GPUs. CUDA includes C/C++ software- development tools, function libraries, and a hardware- abstraction mechanism that hides the GPU hardware from developers. High-performance computing on GPUs has attracted an enthusiastic following in the academic community enabling the use of heterogeneous systems (i.e., CPU+GPU) where CPU & GPU are separate devices with separate DRAMs [13]. The real breakthrough is that CUDA can be implemented on most off-the-shelf Nvidia video cards installed in most personal computers and therefore available to researchers as well as private and public medical practices.

An important feature of CUDA is that application programmers do not write threaded code explicitly. A hardware thread manager handles parallelism automatically, a vital property when multithreading scales to thousands of threads. Although

CUDA automates thread management, it does not entirely relieve developers from thinking about threads. Developers must analyze their algorithms to determine how best to divide the data into smaller chunks for distribution among the thread processors. This data layout or "decomposition" does require programmers to find the optimal numbers of threads and blocks that will keep the GPU fully utilized.

Downsides are few. Mainly, GPUs only recently became fully programmable devices, so their programming interfaces and tools are somewhat immature. Moreover, single-precision floating point is sufficient for consumer graphics, so GPUs do not yet support double precision.

Parallel portions of an application are executed on the device as kernels. They are functions directly called from the CPU that run on the GPU. One kernel is executed at a time and it is - by CUDA using the abstractions of threads, blocks and grids. The GPU chip is organized as a collection of multiprocessors (MPs), with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple MPs and each MP is further divided into a number of stream processors (SPs) which handle one or more threads in a block. Main CUDA's abstraction components can be summarized as follow:

- *Thread*: The atomic execution step, managed by an SP. Each thread uses a local index to access global data elements such that the collection of all threads cooperatively processes the entire data set. The amount of threads is limited to 512 due to the small number of registers that can be allocated across all the threads running in all the blocks assigned to an MP.
- *Block*: A group of threads, handled by the MP. Their execution within a block is completely masked and they could execute concurrently or serially and in no particular order.
- *Grid*: A group of blocks, entirely handled by a single GPU chip. There's no synchronization among blocks.

CUDA software is compatible with several Nvidia's graphic adapters, and it scales autonomously to perfectly fit the amount of MPs available on each GPU. However, in order to improve parallelism efficiency, it is mandatory to keep the GPU multiprocessors equally busy, splitting the computation to maximize the amount of thread and blocks, and concurrently keeping resource usage low enough to support multiple active thread blocks per multiprocessor. In fact, increasing occupancy itself does not necessarily increase performance, thus, to obtain a good trade off it is necessary to parametrize the application through the two levels of parallelization.

B. Acceleration

This section describes how the implementation of the classification algorithm proposed in section II-C can be accelerated using CUDA programming. To obtain an effective increment of computation performances, we need to reduce the iteration loops, which are usually CPU intensive tasks. Fortunately, because loops iterations work on sub-portion of the memory structure, they are quite easy to be parallelized; each parallelized iteration has to be designed to work on a memory segment independent from the ones of the other iterations. In this way the elaboration can be done in parallel without affecting the reliability of the final result.

Algorithm 3 CUDA MMS algorithm

```
1:  MMS (n_rows, n_genes, GEGMatrix, CRCVect)
2:  dimBlock = f(n_gpu, n_threads)
3:  mms = 0
4:  for r=0 to n_rows-1 do
5:     generate BLOCKS of (n_genes / dimBlock)
6:     for all block in BLOCKS do
7:        elaborate MMS contribute of genes in block
8:        update mms with block contributes
9:     end for
10: end for
11: return mms
```

Looking at the algorithms described in Section II-C, we can notice that the final result is composed of different contributions, one for each row. There is no interaction between rows, hence, each contribution can be evaluated separately and in parallel. Figure 2 provides the work-flow structure of our approach to the GEG algorithms. In the first step, the original GEG memory structure is copied from the system memory to the GPU memory. This is part of the CUDA initialization and it allows to pass the whole classification effort from the main system to the GPU system. Once the GPU memory is initialized, Kernel 1 extracts, one by one, each row of the GEG which represents the minimum subset of the GEG structure identified for the actual parallel elaboration. The Kernel 1 instantiates a second kernel (Kernel 2) for each row. Kernel 2 performs the partition of each row in blocks, each referring to a subset of gene expression values. Because each gene contribution is computed in relationship to its successors, a set of threads is defined: for a single gene, a thread elaborates the relation with the successors (the function F). Once a thread ends, it returns the result to Kernel 2 which reconstructs the final result to Kernel 1. All threads are defined among MPs and SPs according to the CUDA architecture.

This approach is summarized in Algorithm 3 for the MMS algorithm. Lines 2 and 3 represent the Kernel 1 initialization of the process by preparing the MMS elaboration (line 3) and extracting each single row (line 4). Once Kernel 2 receives the row, it generates the required blocks (line 5). This step is directly influenced by the number of GPUs available and the number of threads handled by MPs (line 2), as explained in the previous section. Each stream processor executes lines 6 and 7 for a given element of the block. Line 7 implements lines 5 and 6 of the original MMS algorithm (Algorithm 1) in which genes are set with the element associated to each thread. Finally, line 8 is performed by Kernel 1 to account for each block contribute.

The same approach used for the MMS has been used to write the parallel version of the SMS algorithm. In both

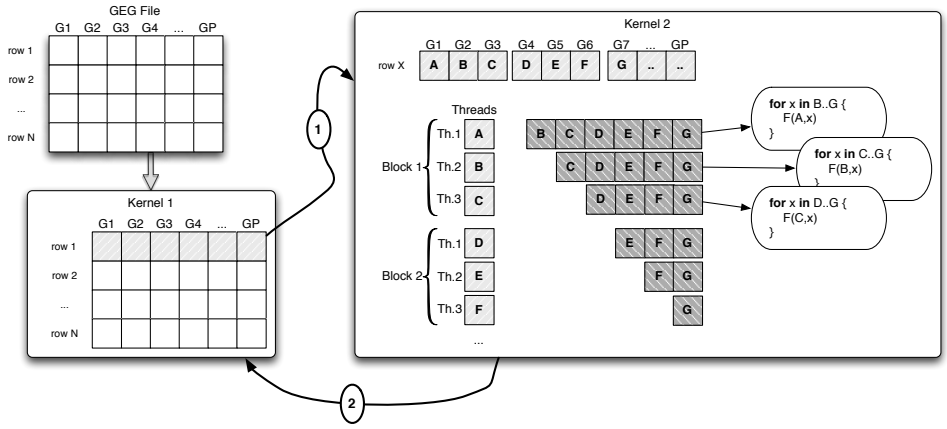

Figure 2. CUDA GPU Workflow

cases, the CUDA architecture allows to suppress the loop for single gene contribution by defining a parallel elaboration of all of them. The scalability of the parallel approach is easily identified by the number of blocks which can be defined, according to the video card architecture.

IV. Results

This section shows the results of the comparison between the standard GEG implementation and its parallel version, and also analyzes how the improvement depends on the code parametrization. The comparison between the two different implementation of the GEG has been calculated for both SMS and MMS.

Tests have been performed on a Quadro FX 1700 by NVidia video card (512MB Memory Interface, 128-bit Graphic Memory Bandwidth, 12.8 GB/sec. Graphics Bus, 32 CUDA Parallel Processor Cores) [14]. The training sets used to build the different GEGs come from experiments run on microarray chips of different sizes: 9K (9216 spots), 18K (18432 spots), 24K (24168 spots), 37K, and 45K (43196 spots) [11].

Figure 3 shows how the parallel implementation is always faster than the standard version. The increase in performance is quite constant among all the training classes, with an average improvement of 50% w.r.t. the original version executed on the same machine. Moreover, the similar improvement among the different training sets also demonstrates how the complexity and size of the GEG does not influence the advantages of the parallelization.

The experimental data is further detailed in Table I where we used different numbers of threads in the CUDA-based computation of the MMS. These additional experiments were run to study the trade-off among several combinations of number of blocks and threads, which follow this relationship: $\#Blocks = rowLength/\#Threads$. From the results shown in the table it is clear how an increase of the number of threads does not always corresponds to an increase in performances. In our experiments, any number of threads greater than 30 resulted in very similar performance improvements. This result demonstrates how, globally, the GEG parallelization is well

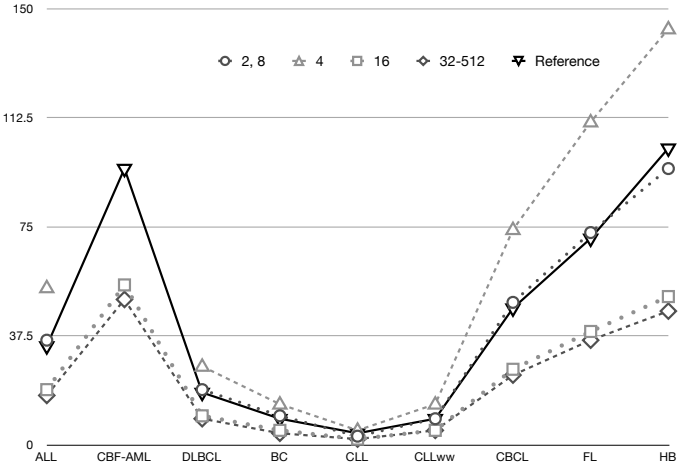

Figure 3. MMS results

supported by the scaling of the CUDA's architecture; the presence of both long (first rows in Fig.2) and short (latest rows in Fig.2) threads keeps the calculation effort well balanced, no matter the amount of blocks. Only when the number of threads falls below 30, then the performance starts to decrease, because not all MPs are completely allocated. In the extreme situation of 2 to 8 threads, performances are in most cases even worse than the original serial version of the algorithm. CUDA architecture generally requires a fine tuning of both thread an block parameters in order to reach its maximum parallelism. The ability to correctly manage GEGs in comparable time intervals, even in different configurations, makes it possible to completely generalize the software's acceleration, no matter the specific settings of the considered device.

As a last comment we can note that a 50% improvement could be also seen as a poor result. Nevertheless, it is necessary to consider that in its current version the parallelization is focused on a quite small portion of the code. The application of the same parallelization to the remaining loops of the algorithms proposed in section section II-C would lead to much higher performance improvements.

GEGs	Number of Threads									Original	Best Saving
	2	4	8	16	32	64	128	256	512		
ALL	36	54	36	19	18	**17**	17	18	18	34	50%
AML	na	na	na	55	50	50	**49**	49	49	95	48.5%
DLBCL	19	27	18	10	9	**9**	9	9	9	18	50%
SBT	10	14	10	5	5	**4**	5	5	5	9	56%
CLL	3	5	4	2	2	**2**	2	2	2	4	50%
CLLww	9	14	10	5	5	5	**4**	5	5	9	55.5%
CBCL	49	74	49	26	24	**24**	24	24	25	47	49%
FL	73	111	73	39	37	**36**	36	36	36	71	49.5%
HB	95	143	96	51	47	**46**	47	47	47	102	55%
TOTAL	294	442	446	212	197	**193**	193	195	196	389	50.5%

* Time expressed in seconds

Table I
TIME RELATIONS

V. CONCLUSION

The parallel processing conversion of the GEG classification algorithm proposed in this paper aims at showing how the parallel distribution of tasks on GPU's dedicated cores heavily improves time performances. Due to its data structure, the GEG classification algorithm is a very good example of how low-cost graphic processor can be used in massive-calculation algorithms. The time performances show an overall time saving of 50% and, surprisingly, a low correlation with any parametrization of the parallelization architecture. Current work is focused on defining even better parallelization strategies to obtain further performance improvements.

VI. ACKNOWLEDGMENTS

The authors wish to acknowledge and thank Alessio Vercellone and Alessandro Morabito, because without their hard work and help this work could not have been completed and submitted in time.

REFERENCES

[1] G. Gibson, "Microarray analysis," *PLoS Biology*, vol. 1, no. 1, pp. 28–29, Oct. 2003.

[2] P. Larranaga, B. Calvo, R. Santana, C. Bielza, J. Galdiano, I. Inza, J. A. Lozano, R. Armananzas, A. Santafe, G. ad Perez, and V. Robles, "Machine learning in bioinformatics," *Briefings in Bioinformatics*, vol. 7, no. 1, pp. 86–112, Feb. 2006.

[3] A. Statnikov, C. F. Aliferis, I. Tsamardinos, D. Hardin, and S. Levy, "A comprehensive evaluation of multicategory classification methods for microarray gene expression cancer diagnosis." *Bioinformatics*, vol. 21, no. 5, pp. 631–643, Mar 2005.

[4] S. Deegalla and H. Boström, "Classification of microarrays with knn: Comparison of dimensionality reduction methods," in *LNCS: Intelligent Data Engineering and Automated Learning (IDEAL)*, vol. 4881, 2007, pp. 800–809.

[5] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[6] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, May 2008, pp. 836–838.

[7] B. Coutinho, G. Teodoro, R. Oliveira, D. Neto, and R. Ferreira, "Profiling general purpose gpu applications," in *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, Oct. 2009, pp. 11–18.

[8] A. Benso, S. Di Carlo, G. Politano, and L. Sterpone, "Differential gene expression graphs: A data structure for classification in dna microarrays," in *8th IEEE International Conference on BioInformatics and BioEngineering (BIBE)*, Oct. 2008, pp. 1–6.

[9] D. B. Allison, X. Cui, G. P. Page, and M. Sabripour, "Microarray data analysis: from disarray to consolidation to consensus," *Nature Reviews: Genetics*, vol. 7, no. 1, pp. 55–65, May 2006.

[10] A. Benso, S. Di Carlo, G. Politano, and L. Sterpone, "A graph-based representation of gene expression profiles in dna microarrays," in *IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, Sept. 2008, pp. 75–82.

[11] cdna stanford's microarray database. [Online]. Available: http://genome-www.stanford.edu/

[12] Cuda technical specifications. [Online]. Available: http://www.nvidia.com/object/cuda_home.html

[13] S. Wei-dong and M. Zong-min, "High-throughput sequence translation using cuda," in *Biomedical Engineering and Informatics, 2009. BMEI '09. 2nd International Conference on*, Oct. 2009, pp. 1–5.

[14] Quadro fx - technical specifications. [Online]. Available: http://www.nvidia.com/page/qfx_mr.html