GPU cards as a low cost solution for efficient and fast classification of high dimensional gene expression datasets

GPU cards as a low cost solution for efficient and fast classification of high dimensional gene expression datasets

A. Benso * **S. Di Carlo** * **G. Politano** * **A. Savino** * **A. Scionti** *

* *Politecnico di Torino, Control and Computer Engineering Department, Corso Duca degli Abruzzi 24, 10129, Torino, Italy (e-mail: firstname.lastname@polito.it).*

Abstract: The days when bioinformatics tools will be so reliable to become a standard aid in routine clinical diagnostics are getting very close. However, it is important to remember that the more complex and advanced bioinformatics tools become, the more performances are required by the computing platforms. Unfortunately, the cost of High Performance Computing (HPC) platforms is still prohibitive for both public and private medical practices. Therefore, to promote and facilitate the use of bioinformatics tools it is important to identify low-cost parallel computing solutions. This paper presents a successful experience in using the parallel processing capabilities of Graphical Processing Units (GPU) to speed up classification of gene expression profiles. 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 bioinformatics tools and real medical practice.

Keywords: bioinformatics, pattern recognition, parallel computing, software performance.

1. 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, this vast amount of data requires both advanced specialized bioinformatics tools and powerful computing platforms. A common characteristic to several bioinformatics applications is the high data dimensionality and the repetitive execution of heavy computational cycles.

Analysis and classification of gene expression profiles from DNA microarrays is a typical example of a fundamental task in bioinformatics. 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 (Gibson, 2003; Larranaga et al., 2006).

Microarray data analysis and classification involves the analysis and correlation of thousands of variables. It should be further considered that future technologies continuously increase the amount of information provided by these supports (the new Affymetrix GeneChip^{TM}array maps 500K probes on a single device). Special-purpose hardware, as for instance clusters of computers or Field-Programmable Gate Arrays (FPGA) are interesting options to elaborate this huge amount of data. However, they tend to be very expensive and not largely available to many users. For these reasons, in statistical classification of gene expression data, most available classifiers need strong data dimensionality reduction to make the computation complexity affordable by available general purpose computational platforms (Statnikov et al., 2005; Deegalla and Boström, 2007). This, sometimes, results in reduced accuracy of the classification model due to the loss of relevant information.

Recently, a new classification algorithm for gene expression profiles able to work with high dimensionality data has been proposed in (Benso et al., 2008, 2010). The algorithm represents gene expression profiles by means of Gene Expression Graphs (GEG) and performs the classification by comparing graphs representing different phenotypes. While it partially overcomes the problems of other statistical classifiers when managing high dimensional data, its high computation time still represent a major problem for its widespread adoption.

This paper investigates the use of off-the-shelf Graphic Processor Units (GPUs) to accelerate the GEG-based classification algorithm proposed in (Benso et al., 2008, 2010). The availability of such an accelerated classifier has the potentiality of avoiding data dimensionality reduction during gene expression profiles analysis while keeping the computation time under control. This minimizes the probability of discarding information potentially useful for the analysis, allowing to obtain maximum accuracy from the classification model.

2. PARALLEL COMPUTING AND GPU

Parallel computing is a very attractive solution to speed up advanced bioinformatics tools. It allows many calcula-

tions to be carried out simultaneously by dividing large problems into smaller ones solved concurrently. According to the level at which the hardware supports parallelism, parallel computers can be roughly classified into: (i) *multi-core* and *multi-processor computers* having multiple processing elements within a single machine; (ii) *distributed computers* including clusters, Massive Parallel Processing (MPP) computers and grid computers using multiple machines connected through a network; (iii) *specialized parallel computers* including reconfigurable computing platforms based on FPGAs and GPUs (Bader and Pennington, 2001; Fan et al., 2004). Each of these classes involves different costs. Figure 1 gives a general summary of how the cost of a gigaflop decreased in the last 10 years depending on different architectures.

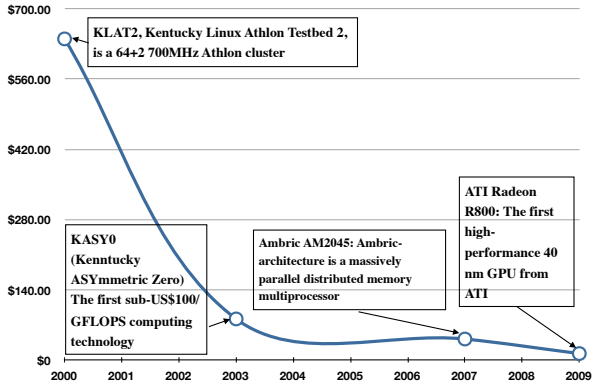

Fig. 1. Economic trend of GFLOP cost in the last 10 years.

The use of GPUs for parallel processing is a recent paradigm that turns the massive floating-point computational power of a modern graphics accelerator into a general-purpose computing platform. Today, parallel GPUs have begun making computational inroads against general-purpose CPUs, and a subfield of research, dubbed GPGPU for General Purpose Computing on GPU, has found its way into diverse fields (Nickolls et al., 2008; Luebke, 2008; Coutinho et al., 2009; Wei-dong and Zongmin, 2009). Beyond their appeal as cost-effective high-performance computing accelerators, GPUs also significantly reduce space, power, and cooling demands compared to other parallel platforms. For instance, compared to the latest quad-core CPUs, NVIDIA Tesla C2050$^{\mathrm{TM}}$and C2070$^{\mathrm{TM}}$processors deliver equivalent computing performances at 1/10th the cost and 1/20th the power consumption. GPUs are therefore one of the most viable solutions to deliver low-cost and easily available parallel computation to bioinformatics applications (Manavski and Valle, 2008).

2.1 The CUDA programming model

CUDA (*Compute Unified Device Architecture*) is an extension of the C programming language for massive parallel high-performance computing on the NVIDIA's GPUs (NVIDIA, 2010a). It includes C/C++ software-development tools, function libraries and a hardware-abstraction mechanism that hides the GPU hardware from developers thus overcoming the complexity of previous GPGPU approaches. The real breakthrough is that CUDA can be used on several off-the-shelf NVIDIA video cards installed in most personal computers already available to researchers as well as private and public scientific institutions.

In CUDA, the GPU is viewed as a compute device suitable for parallel data applications. It has its own random access memory and may be used to run *kernels*, i.e., functions directly called from the CPU of the computer hosting the GPU. One kernel at a time can be parallelized on the GPU by splitting it into *threads*. Threads are grouped into *blocks* and many *blocks* may run in a *grid* of blocks. The GPU contains a collection of multiprocessors (MPs) each one responsible for handling one or more blocks of a grid. A block is never divided across multiple MPs. Each MP is further divided into a number of stream processors (SPs) each one handling one or more threads of a block. Threads of the same block share data through fast shared on-chip memory and can be synchronized through synchronization points.

An important feature of CUDA is that 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 threads management, it does not entirely relieve developers from thinking about threads. Developers must analyze their algorithms to determine the best way to divide data into smaller chunks for distribution among the thread processors and for optimal use of the available memory. 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.

3. GEG-BASED CLASSIFIER

The GEG-based classifier proposed in (Benso et al., 2008, 2010) performs classification by representing gene expression profiles as Gene Expression Graphs (GEG). GEGs are constructed from raw gene expression measures obtained from the scanning and preprocessing process of microarray experiments (Gibson, 2003; Allison et al., 2006). GEGs model groups of gene expression profiles with common characteristics (e.g., same disease) in a single structure. The model is constructed in order to allow efficient classification, to avoid the influence of pre-processing steps on the prediction process, and to work with minimum dimensionality reduction on the raw data.

In particular, a set S of samples obtained from different microarray experiments can be modeled by a non-oriented weighted graph $GEG = (V, E)$ where:

- each vertex $v_x \in V$ represents a gene. Only vertices representing *relevant* genes are included in the graph;
- each edge $(u, v) \in E \subseteq V \times V$ 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 $(u,v) \in E$ 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.

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:

$$CRC_v = \sum_{\forall s \in S} Rel_v(s) \tag{1}$$

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

GEGs are 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 set of training samples Tr_{pat}, and one representing the sample $\boldsymbol{s}$ 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 among 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.

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

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

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

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

$$Z_x = CRC_{x_{pat}} \cdot CRC_{x_s} \tag{4}$$

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-term 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 vice versa;
- $Z_x = 0$: 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-term of each gene always positive.

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

In a classification experiment the GEG representing a given sample is compared to different GEGs each corresponding to one of the phenotypes included in the classification library. The comparison returning the maximum proximity score can be used to identify the phenotype of the sample.

3.1 Classifier implementation

Since the GEG-based classifier compares 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 one composed of several thousands of vertices (genes). In order to reduce the required amount of memory GEGs are never explicitly represented with an adjacency matrix or list. Instead, we use a matrix with rows associated to experiments used to build the GEG and columns associated to genes. Each element contains the relevance of the gene in the considered experiment. Therefore the actual software representation of a GEG is a matrix of experiments and their gene relevance values named here *GEGMatrix*.

Alg. 1 shows the MMS computation algorithm. For each row of the GEGMatrix (lines 3-13) the MMS contribution of each gene is computed (line 4) w.r.t. the set of the remaining genes (lines 5-11).

Algorithm 1 MMS computation algorithm

```
 1: MMS (n_rows, n_genes, GEGMatrix, CRCPat)
 2: mms = 0
 3: for r=0 to n_rows-1 do
 4:    for i=0 to n_genes-2 do
 5:       for j=i+1 to n_genes - 1 do
 6:          if (GEGMatrix[r][i]*GEGMatrix[r][j]) != 0 then
 7:             num = pow(CRCPat[i],2) + pow(CRCPat[j],2)
 8:             den = abs(CRCPat[i]) + abs(CRCPat[j])
 9:             mms += num/den
10:          end if
11:       end for
12:    end for
13: end for
14: return mms
```

In a similar way the SMS is computed according to Alg. 2. The matrix exploration remains the same, and the CRCSample and CRCPat vectors storing the cumulative relevance counts of both GEG_{pat} and GEG_s are used to compute each SMS contribution according to eq. 3.

Both Alg. 1 and Alg. 2 perfectly match two critical requirements for an efficient implementation with CUDA: first, they deal with massive floating point operations, and second they perform repetitive arithmetical operations among identically structured data.

Algorithm 2 SMS computation algorithm

```
1: SMS ( n_rows, n_genes, GEGMatrix, CRCPat, CRCSample)
2: sms = 0
3: for i=0 to n_genes-2 do
4:    if CRCSample[i] != 0 then
5:       Zi = CRCPat[i]*CRCSample[i]
6:       for j=i+1 to n_genes - 1 do
7:          if CRCSample[j] != 0 then
8:             Zj = CRCPat[j]*CRCSample[j]
9:             num = ( Zi * abs(Zi) ) + ( Zj * abs(Zj) )
10:            den = abs(Zi) + abs(Zj)
11:            for r= 0 to n_rows -1 do
12:               if (GEGMatrix[r][i]*GEGMatrix[r][j]) == 1 then
13:                  sms += num/den
14:               end if
15:            end for
16:         end if
17:      end for
18:   end if
19: end for
20: return sms
```

4. GEG-BASED CLASSIFIER IMPLEMENTATION WITH CUDA

This section describes how the implementation of the GEG-based classifier proposed in section 3.1 can be accelerated using CUDA. To obtain a significant increment of the computation performances the best practice is to try to parallelize as much as possible time consuming loop iterations. Fortunately both Alg. 1 and Alg. 2 include iterations that work on sub-portions of the GEGMatrix. They can be therefore easily parallelized. Each parallelized iteration has to be designed to work on a memory portion of GEGMatrix independent from the ones of the other iterations. In this way the elaboration can be done in parallel without affecting the correctness of the final result.

Looking at Alg. 1 it is clear that the final result is composed of different contributions: one for each row of GEGMatrix, and for each row one contribution for each gene. Alg. 3 proposes a parallel implementation of Alg. 1. The proposed implementation is based on two kernels.

The elaboration is performed iterating on each row (line 4). Kernel-1 (called at line 8) is in charge of copying one row of GEGMatrix into a reserved area into the the GPU memory (gpu_row). This is part of the CUDA initialization process. It allows to pass the whole classification effort from the main system's CPU to the GPU. The notation <<< dimBlock, dimGrid >>> indicates that this kernel is parallelized by considering a grid of dimGrid blocks with each block composed of dimBlock threads. In this case the copy of the different elements of the row is performed in parallel to speed up the process. Each generated thread executes the function cudaMMS_k1 in parallel (lines 18-20). Each thread is identified by a block identifier (blockIdx) and by a thread identifier (threadIdx) used to compute the element to be copied (line 19). The cudaThreadSynchronize function (line 9) allows to synchronize the different threads in order to wait the end of the copy before moving to the next step of the computation. The calculation of the MMS is computed again in parallel through the kernel cudaMMS_k2 (lines 22-30). Each instance of this kernel (thread) analyzes the gene idx of the row and compares it to all following genes in the row according to the original algorithm Alg. 1 (lines 24-29). All calculations are performed on the copy of the row stored in the GPU's memory, thus allowing fast access to this information. When all threads computing the partial contributions of the MMS are completed (line 11), the single contributions are transferred from the GPU memory to the CPU memory (line 12) and the contribution of the row to the MMS is reconstructed by adding the single contributions (lines 13-15).

Algorithm 3 CUDA MMS algorithm

```
1: MMS (n_rows, n_genes, GEGMatrix, CRCPat)
2: mms = 0
3: define dimBlock, dimGrid according to CUDA architecture
4: for r=0 to n_rows-1 do
5:    cpumalloc (cpu_mms[n_genes])
6:    cudamalloc (gpu_mms[n_genes])
7:    cudamalloc (gpu_row[n_genes])
8:    cudaMMS_k1 <<< dimBlock, dimGrid >>> (r, n_genes,
      GEGMatrix, gpu_row)
9:    cudaThreadSynchronize
10:   cudaMMS_k2 <<< dimBlock, dimGrid >>>(n_genes, CR-
      CPat, gpu_row, gpu_mms)
11:   cudaThreadSynchronize
12:   cudamemcopy (cpu_mms,gpu_mms)
13:   for i=0 to n_genes-1 do
14:      mms+=cpu_mms[i]
15:   end for
16: end for
17: ===== Kernel-1 (threaded)
18: cudaMMS_k1 (row, n_genes, GEGMatrix, gpu_row)
19: idx = blockIdx * dimBlock + threadIdx;
20: gpu_row[idx] <= GEGMatrix[row*n_genes+idx]
21: ===== Kernel-2 (threaded)
22: cudaMMS_k2 (n_genes, CRCVect, gpu_row, gpu_mms)
23: idx = blockIdx * dimBlock + threadIdx;
24: for j=idx+1 to n_genes - 1 do
25:    if (gpu_row[idx] * gpu_row[j]) != 0 then
26:       num = pow(CRCVect[idx],2) + pow(CRCVect[j],2)
27:       den = abs(CRCVect[idx]) + abs(CRCVect[j])
28:       gpu_mms[idx] += num/den
29:    end if
30: end for
```

Figure 2 graphically shows how threads are generated during the execution of Alg. 3.

The computation of the SMS works in a slightly different way since it is performed in two steps: the first one calculates the weigh of each edge of GEG_{pat} connecting expressed genes (Alg. 4), while the second computes the final SMS (Alg. 5). In Alg. 4, as in the previous one, all required memory structures are allocated, in both the GPU (lines 4, 5) and the main system's CPU (line 6). Being the GEG an undirected simple weighted graph with no self-loops, its adjacency matrix is symmetric and its diagonal is a zero-set. Consequently, the only useful part of the matrix is its strictly upper subset that in the code has been implemented as an array (lines 3, 4). This solution saves a considerable amount of memory which is a key requirement in GPU programming but involves a more complex indexing. The *getAdjacencyArrayIndex* function (line 24) performs the index calculation according to the array representation. Kernel-1 (called at line 8) is demanded to copy each selected row to the proper GPU memory structure. Kernel-2 (called at line 10), performs the weight computation for all pairs of genes

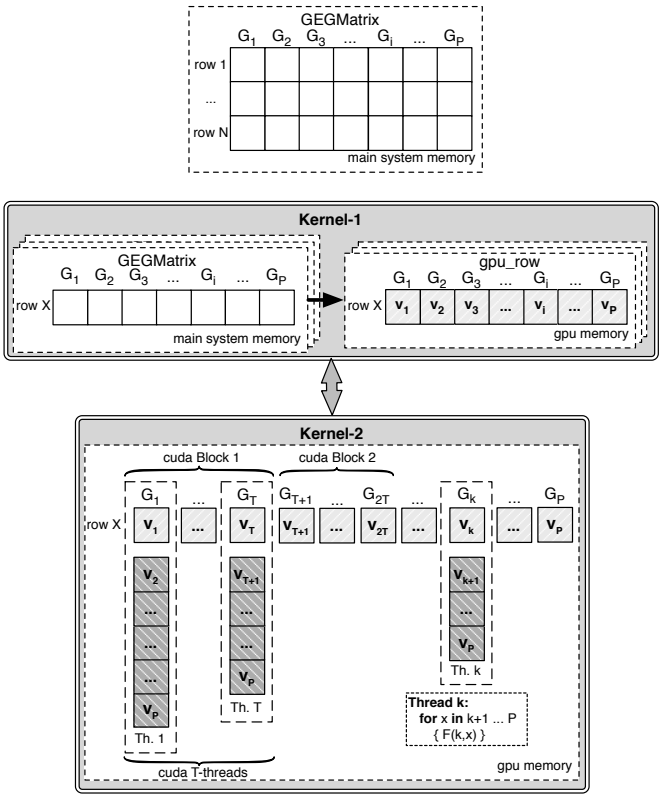

Fig. 2. CUDA GPU Workflow.

Algorithm 4 CUDA vector of weights computation algorithm

```
1: WijFunct (n_rows, n_genes, GEGMatrix)
2: define dimBlock, dimGrid according to CUDA architecture
3: adjacency_matrix_dim = ( n_genes * (n_genes - 1) ) / 2
4: cudamalloc (gpu_wij[adjacency_matrix_dim])
5: cudamalloc (gpu_row[n_genes])
6: cpumalloc (cpu_wij[adjacency_matrix_dim])
7: for r=0 to n_rows-1 do
8:    cudaWij_k1 <<< dimBlock, dimGrid >>> (r, n_genes,
      GEGMatrix, gpu_row);
9:    cudaThreadSynchronize();
10:   cudaWij_k2 <<< dimBlock, dimGrid >>> (gpu_row,
      n_genes, gpu_wij);
11:   cudaThreadSynchronize();
12: end for
13: cudamemcopy (cpu_wij, gpu_wij)
14: return cpu_wij
15: ===== Kernel-1 (threaded)
16: cudaWij_k1 (row, n_genes, GEGMatrix, gpu_row)
17: idx = blockIdx * dimBlock + threadIdx;
18: gpu_row[idx] <= GEGMatrix[row*n_genes+idx]
19: ===== Kernel-2 (threaded)
20: cudaWij_k2 (gpu_row, n_genes, gpu_wij)
21: idx = blockIdx * dimBlock + threadIdx;
22: for j=idx+1 to n_genes - 1 do
23:    if ((abs(gpu_row[idx]) == 1) AND (abs(gpu_row[j]]) == 1))
      then
24:       adj_index = getAdjacentArrayIndex(idx, j)
25:       gpu_wij[adj_index] ++
26:    end if
27: end for
```

in the row. The two kernels are synchronized by the cudaThreadSynchronize function previously detailed. At completion of all generated threads, the GPU weight memory structure is copied back into the host memory (line 13). Alg. 5 computes a similar set of operations as the one of Alg. 1, with the difference that the sample under test is taken into account by Kernel-1 and Kernel-2 (lines 9 and 11). It causes that the gene contribution is evaluated only if it is present in the sample under test (line 25).

In order to fit the requirements of each specific problem, algorithms have to properly manage the GPU blocks and grids definition. In the proposed implementation, the algorithm uses the following definitions: $dimBlock = \#Threads$, $dimGrid = (n\_genes/dimBlock) + 1$. This solution allows to balance the computational effort depending on the number of genes. In the proposed implementation, best results are obtained setting to 64, 64 and 470, the number of Threads, the dimBlock and dimGrid values respectively, giving an average of 30K genes.

5. RESULTS

This section presents the results of a set of experiments run to demonstrate the effectiveness of the proposed approach. Results show the improvement in the computation of the different components of the Proximity Score introduce in eq. 2: MMS, SMS (including the computation of the different weights w_{ij}). All tests have been performed on a NVIDIA Quadro FX 1700 video card (512MB Memory Interface, 128-bit Graphic Memory Bandwidth, 12.8 GB/sec. Graphics Bus, 32 CUDA Parallel Processor Cores) NVIDIA (2010b). The training sets used to build the different GEGs and to train the classifier come from experiments run on microarray chips of different sizes: 9K (9216 spots), 18K (18432 spots), 24K (24168 spots), 37K, and 45K (43196 spots) (Stanford, 2010). The data set used to test the parallelized classification is instead composed of two different subsets of microarray experiments. One subset of samples belongs to the set of diseases included in the classification library and forms a set of classifiable samples (In-class or INC samples). The second subset is a test set of unclassifiable samples (out-of-class or OOC samples), i.e., samples that do not belong to any of the class in the classification library. These samples are used to better analyze the classification when samples need to be rejected. Table 1 summarizes the results for the parallelization of the MMS computation algorithm. To study the trade-off and the sensitivity to the parallel algorithm parametrization, all experiments have been repeated with different number of threads. The first important result is that an average 50% improvement can be obtained in the computation of the MMS for all training GEGs, and this improvement is independent of their size. A second important consideration is that an increase of the number of threads does not always correspond to an increase in performances. In all the experiments, any number of threads greater than 30 resulted in very similar performance improvements, again regardless of the GEGs size. Only when the number of threads falls below 30, the performances decrease because not all MPs are completely assigned. In the extreme situation of 2 to 8 threads, performances are in most cases even worse than the original serial version of the algorithm. This result is interesting because it suggests that, even if the CUDA architecture usually requires a fine tuning of both thread and block parameters in order to reach its maximum parallelism, in the case of the parallel

Algorithm 5 CUDA SMS algorithm

```
1: SMS (n_rows, n_genes, CRCSample, CRCPat)
2: sms = 0
3: define dimBlock, dimGrid according to CUDA architecture
4: for r=0 to n_rows-1 do
5:     cpumalloc (cpu_sms[n_genes])
6:     cudamalloc (gpu_sms[n_genes])
7:     cudamalloc (gpu_row[n_genes])
8:     cudaSMS_k1 <<< dimBlock, dimGrid >>> (r, n_genes,
       CRCSample, gpu_row)
9:     cudaThreadSynchronize
10:    cudaSMS_k2 <<< dimBlock, dimGrid >>> (n_genes, CR-
       CPat, gpu_row, gpu_wij, gpu_sms)
11:    cudaThreadSynchronize
12:    cudamemcopy (cpu_sms,gpu_sms)
13:    for i=0 to n_genes-1 do
14:        sms+=cpu_sms[i]
15:    end for
16: end for
17: ===== Kernel-1 (threaded)
18: cudaSMS_k1 (row, n_genes, CRCSample, gpu_row)
19: idx = blockIdx * dimBlock + threadIdx;
20: gpu_row[idx] <= CRCSample[row*n_genes+idx]
21: ===== Kernel-2 (threaded)
22: cudaSMS_k2 (n_genes, CRCPat, gpu_row, gpu_wij,
    gpu_sms)
23: idx = blockIdx * dimBlock + threadIdx;
24: if gpu_row[idx] != 0 then
25:    Zi = CRCPat[idx] * gpu_row[idx]
26:    for j=idx+1 to n_genes - 1 do
27:        if gpu_row[j] != 0 then
28:            Zj = CRCPat[j] * gpu_row[j]
29:            num = ( Zi * abs(Zi) ) + ( Zj * abs(Zj) )
30:            den = abs(Zi) + abs(Zj)
31:            sms += gpu_wij[j] * num/den
32:        end if
33:    end for
34: end if
```

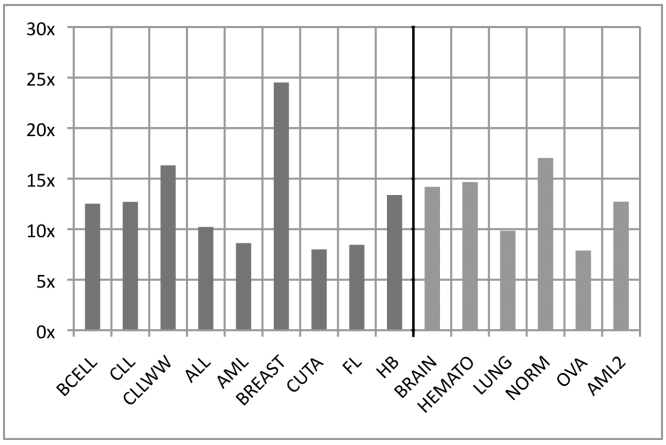

Fig. 3. SMS improvement including the computation of the different weights

MMS algorithm the maximum possible improvement can be reached using about 30 threads, regardless the size of the GEG, making the tuning not necessary.

Figure 3 shows instead the improvement obtained for the SMS computation including the computation of the different weights. The figure is divided in two parts, showing the improvement when comparing the GEGs representing the

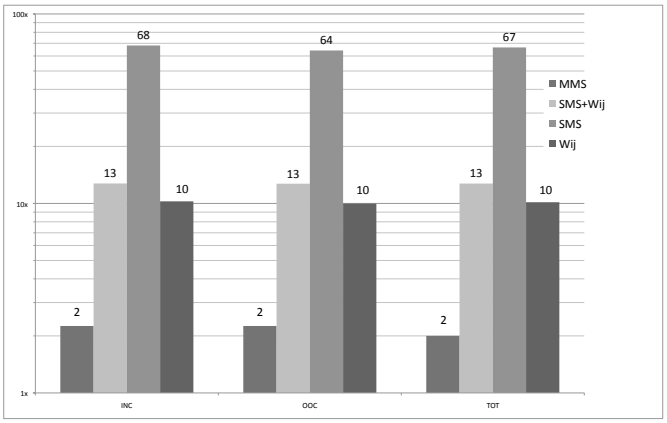

Fig. 4. Average improvement for each Algorithm

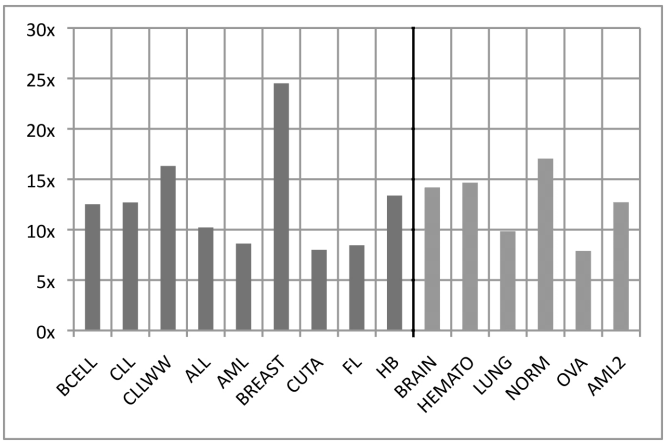

Fig. 5. Total time improvement (MMS+Wij*SMS)

available classes with the INC samples of the test set (right side), and the OOC samples (right side), respectively. The results for the SMS computation are even more promising than for the MMS, reaching an average improvement between 5 to 25 times the original computation time. Figure 4 breaks down the contribution to the total improvement of the three implemented algorithms (MMS, SMS, and weights). The first important observation is the general robustness of the implementation against OOC and INC samples classification. This suggests that all algorithms have a solid behavior when the comparison is done with both a wide or small gene intersection between the two compared GEGs. From the figure it also emerges the huge acceleration obtained in the weights computation (66x); unfortunately the contribution of the weights computation in the total computation time of the SMS is very small and therefore the improvement in its computation is not heavily reflected in the overall improvement. It nevertheless allows to speed-up the total computation of the SMS of an average of 13x. The average MMS improvement is assessed at about 2x.

Figure 5 presents a summary of the overall average improvement in the Proximity Score computation for both INC and OOC samples. The average trend of about 12x improvement is synthesized by the linear regression fitting line. The lower bound of the acceleration is also not far from the average curve, and it guarantees an average im-

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 1. Time Relations

provement of at least 5x, but with very high upper bounds in all GEGs.

6. CONCLUSION

The parallel processing conversion of the GEG-based classification algorithm proposed in this paper aims at showing how the parallel distribution of tasks on GPU's dedicated cores heavily improves time performances at very reasonable costs. Due to its data structure, the GEG-based classification algorithm is a very good example of how low-cost graphic processors can be used in massive-calculation algorithms. Apart from the scientific contribution of the parallel version of the GEG based classifier, the presented work represents a very interesting result in the definition of low-cost methodologies and architectures for improving the performances of complex bioinformatic algorithms, with the final goal of making them available to different level of clinical practices and not only to a selected number of resourceful laboratories.

ACKNOWLEDGEMENTS

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.

REFERENCES

Allison, D.B., Cui, X., Page, G.P., and Sabripour, M. (2006). Microarray data analysis: from disarray to consolidation to consensus. *Nature Reviews: Genetics*, 7(1), 55–65.

Bader, B. and Pennington, R. (2001). Cluster Computing: Applications. *The International Journal of High Performance Computing*, 15(2), 181–185.

Benso, A., Di Carlo, S., and Politano, G. (2010). A cDNA microarray gene expression data classifier for clinical diagnostics based on graph theory. *ACM/IEEE Transactions on Computational Biology and Bioinformatics*.

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

Coutinho, B., Teodoro, G., Oliveira, R., Neto, D., and Ferreira, R. (2009). Profiling General Purpose GPU Applications. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, 11–18.

Deegalla, S. and Boström, H. (2007). Classification of microarrays with kNN: Comparison of dimensionality reduction methods. In *LNCS: Intelligent Data Engineering and Automated Learning (IDEAL)*, volume 4881, 800–809.

Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. (2004). GPU Cluster for High Performance Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 47.

Gibson, G. (2003). Microarray analysis. *PLoS Biology*, 1(1), 28–29.

Larranaga, P., Calvo, B., Santana, R., Bielza, C., Galdiano, J., Inza, I., Lozano, J.A., Armananzas, R., Santafe, G. ad Perez, A., and Robles, V. (2006). Machine learning in bioinformatics. *Briefings in Bioinformatics*, 7(1), 86–112.

Luebke, D. (2008). CUDA: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, 836–838.

Manavski, S.A. and Valle, G. (2008). CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC bioinformatics*, 9 Suppl 2(Suppl 2), S10+.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable Parallel Programming with CUDA. *Queue*, 6(2), 40–53.

NVIDIA (2010a). CUDA Technical Specifications. URL `http://www.nvidia.com/object/cuda_home.html`.

NVIDIA (2010b). Quadro FX - Technical Specifications. URL `http://www.nvidia.com/page/qfx_mr.html`.

Stanford, U. (2010). cDNA Stanford's Microarray database. URL `http://genome-www.stanford.edu/`.

Statnikov, A., Aliferis, C.F., Tsamardinos, I., Hardin, D., and Levy, S. (2005). A comprehensive evaluation of multicategory classification methods for microarray gene expression cancer diagnosis. *Bioinformatics*, 21(5), 631–643.

Wei-dong, S. and Zong-min, M. (2009). High-Throughput Sequence Translation Using CUDA. In *Biomedical Engineering and Informatics, 2009. BMEI '09. 2nd International Conference on*, 1–5.