Enhancing the Performance of a Microarray Gridding Algorithm via GPU Computing Techniques

Stamos Katsigiannis, Eleni Zacharia, and Dimitris Maroulis, *Member, IEEE*

***Abstract*—cDNA microarrays are a useful tool for studying the expression levels of genes. Nevertheless, microarray image gridding remains a challenging and complex task. Most of the microarray image analysis tools require human intervention, leading to variations of the gene expression results. Automatic methods have also been proposed, but present high computational complexity. In this work, the performance enhancement via GPU computing techniques of a fully automatic gridding method, previously proposed by the authors' research group, is presented. The NVIDIA CUDA architecture was utilized in order to achieve parallel computation of complex steps of the algorithm. Experimental results showed that the proposed approach provides enhanced performance in terms of computational time, while achieving higher utilization of the available computational resources.**

I. Introduction

CDNA microarrays [1] are a powerful biotechnology tool for scientists to simultaneously study the expression levels of thousands of genes. Since their development in 1995, they have been broadly used in many biomedical application areas such as research on cancer, diagnosis and treatments of various diseases, toxicology and pharmacology.

Image analysis is an important aspect of a cDNA microarray experiment as a means for extracting meaningful biological conclusions. The first stage of microarray image analysis is gridding, which is the process of segmenting a microarray image into numerous compartments, each containing one individual spot and background. Gridding is a very challenging process due to the poor quality of microarray images [2][3]: they contain inhomogeneous background and are contaminated with noise and artifacts. Furthermore, microarray images contain thousands of spots of various sizes, shapes, and intensity levels. Spot positions often deviate from their ideal ones - that is 2D array layouts – resulting to misalignments and local deformations of the ideal rectangular grid.

As a result, a number of software programs and techniques have been introduced for gridding. Software packages such as ImaGene [4], ScanAlyze [5] and SpotFinder [6] demand user intervention for adjusting various parameters in order for a predefined grid to match the spots in the microarray image [7]. GenePix and QuantArray software programs automatically adjust the template grid [8][9]. However, they are not efficient when there are misalignments and local deformations of the grid [7]. Other well-known techniques use mathematical morphology [10][11], axis projections [12], markov random field [13] or graphs [14]. A basic drawback of these techniques is that they require human intervention. The absence of a fully automatic procedure for gridding leads to significant discrepancies in the results of the gene expression levels, even for the same microarray slide [15].

In this work, the authors try to enhance the performance of a fully automatic gridding method, previously proposed by their research group [16], which is based on a genetic algorithm approach. This approach can efficiently construct a grid structure in a microarray image even in the presence of noise and artifacts or in the case of various kinds of perturbations such as arbitrary rotations, local deformations and missing spots. Nevertheless, due to the use of a genetic algorithm, this method requires much computational time. In order to tackle this drawback, the use of GPU computing techniques is proposed and evaluated.

Modern personal computers are commonly equipped with powerful graphics processors (GPUs), which are specialized in handling computations for the display of computer graphics. This computational power usually remains underutilized, especially in the case of general computations. General purpose computing on graphics processing units (GPGPU) is the set of techniques that use a GPU in order to perform computations traditionally handled by a CPU. The highly parallel structure of GPUs makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel and in isolation, keeping inter-thread data exchange to a minimum. For this work, the NVIDIA Compute Unified Device Architecture (CUDA) [17][18] has been selected due to the extensive capabilities and optimized API it offers.

The rest of this paper is organized in three sections. Section II describes the gridding method and the proposed GPU approach. Results from the experimental evaluation are presented and discussed in Section III, whereas conclusions and future work are presented in Section IV.

II. Methodology

A. Microarray image gridding algorithm

Our method for gridding microarray images [16] relies on a genetic algorithm (GA) [19][20] which determines the line-segments constituting the borders between two adjacent blocks or spots. The GA is executed twice: First, it

Manuscript received July 27, 2013.

S. Katsigiannis, E. Zacharia and D. Maroulis are with the Real-time Systems and Image Analysis Laboratory, Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, 15703 Athens, Greece (email: stamos@di.uoa.gr; eezacharia@gmail.com; d.maroulis@di.uoa.gr).

978-1-4799-3163-7/13/$31.00 ©2013 IEEE

determines the 'vertical' line-segments of the grid structure and afterwards the 'horizontal' line-segments. The steps of the GA are summarized on Table I.

TABLE I
STEPS OF THE GENETIC ALGORITHM

1. Generate a population of N chromosomes
2. Evaluate the fitness $F(m)$ of each chromosome m.
3. If the termination criterion is not satisfied
   a. Place p_r% of the best chromosomes to the next population
   b. While the size of new population is less than N:
      i. Select four chromosomes (m_1,m_2,m_3,m_4) using the tournament selection operator
      ii. Create four offspring (o_1,o_2,o_3,o_4) by applying the crossover and mutation operators to (m_1,m_2) and (m_3,m_4)
      iii. Evaluate the fitness of o_1,o_2,o_3,o_4
      iv. Place the best two of o_1,o_2,o_3,o_4 to the next population
   c. Replace the current population with the new one. Go to step 2
4. Keep the best chromosome as the solution to the gridding problem

It is worth pointing out that the chromosome m represents all the 'vertical' or 'horizontal' line-segments of a grid structure. However, instead of encoding the variables of all the line-segments, it encodes the variables of one line-segment and the distance *d between two adjacent line-segments.*

Furthermore, each chromosome m is evaluated using a fitness function F(m). The objectives of the gridding procedure are: (i) To maximize the number of line-segments which are determined simultaneously; (ii) To maximize the probabilities of all the determined line segments to be part of the grid.

The probability of a line-segment to be part of the grid is defined by the following equation:

$$P(L_i) = f_B^{R_{L_i}}(L_i) - f_S^{R_{L_i}}(L_i) \quad (1)$$

where R_{Li} denotes the region of the image or block containing the pixels whose distance from the line segment L_i is less than a margin w. The $f_B^{R_{Li}}$, $f_S^{R_{Li}}$ are two real valued functions expressing the percentage of pixels of the region R_{Li} which belong to the background or foreground respectively.

The Fitness Function $F(m)$ of a chromosome m that encodes a possible solution to the particular optimization problem is defined by the following equation:

$$F(m) = \begin{cases} S_p(m) \cdot N(m), \ if \ f_{LS}(m) \le f_{Max} \\ S_p(m), \ otherwise \end{cases} \quad (2)$$

where $S_p(m)$ function denotes the total sum of the probabilities $P(L_i)$ of the line-segments L_i that are represented by the Chromosome m, and have a '*high*' probability $P(L_i)$ to be part of the grid. The $f_{LS}(m)$ function denotes the percentage of the line-segments L_i that are represented by the chromosome m, and have a '*low*' probability $P(L_i)$ to be part of the grid. $P(L_i)$ is considered as '*high*' when its value is higher than a threshold P_{Max}, and is considered as '*low*' when its value is lower than a threshold P_{Low}. For the computation of $S_p(m)$ and $f_{LS}(m)$ each line-segment L_i is labeled by binary values q_i and k_i indicating if $P(L_i)$ is higher or lower than $P_{Max,}$ or P_{Low} respectively.

The gridding results for a block of a microarray image are shown on Fig. 1.

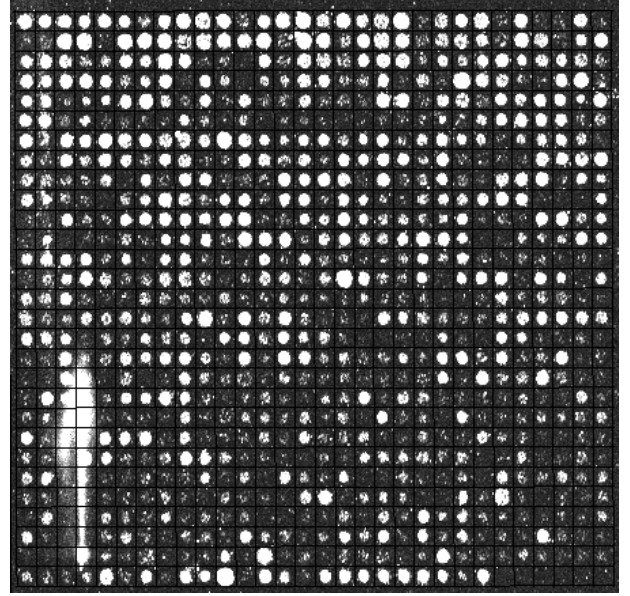

Fig.1. Gridding results for a microarray block.

B. GPU approach

By design, each generation of the chromosome population created by a genetic algorithm requires the execution results of the previous generation. As a result, parallel execution of all the genetic algorithm generations cannot provide enhanced performance, since the computation of each generation would be postponed until all the previous generations have been completed. The performance in that case would be the same as in the case of calculating each generation one by one, plus the overhead from the synchronization operations needed to ensure the computation of each generation in the correct order. Considering this fact, the new implementation should focus on enhancing the performance for computing each generation by exploiting the steps of the algorithm that can be effectively parallelized, keeping inter-thread data exchange and memory transfers to and from the main memory and GPU dedicated memory to a minimum

By examining the aforementioned method, it is evident that the most computationally complex step is the calculation of the fitness $F(m)$ of each chromosome m at every generation. As a consequence, reducing the computation time needed for computing $F(m)$ is expected to provide enhanced performance. In this work, the proposed implementation focuses on transferring the computation of $F(m)$ to the GPU while all the other steps of the algorithm are computed on the CPU.

For the computation of $F(m)$, first the data of all the chromosomes in the population are transferred to the GPU memory in order to avoid multiple memory transfers. Then

the number of the line-segments $N(m)$ is calculated and $N(m)$ CUDA threads are used in order to calculate $P(L_i)$, q_i and k_i, with the i-th thread assigned to the respective measurements. For storing the results, two buffers of size $N(m)$ elements are utilized. On the first buffer the product $P(L_i)\cdot q_i$ is stored on the i-th position, while on the second buffer the binary value k_i is stored on the respective position. An outline of this procedure is shown on Fig. 2.

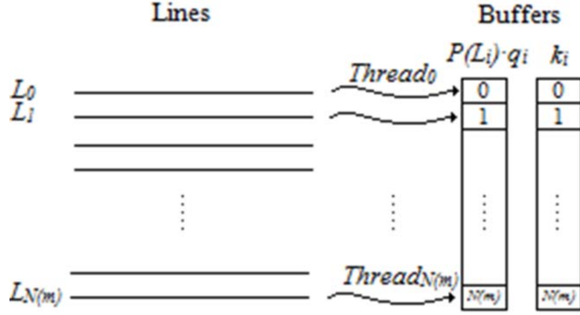

Fig.2. Outline of the GPU implementation for computing the fitness $F(m)$.

Computing the sums $S_p(m)$ and $f_{LS}(m)$ without the use of memory buffers would require thread synchronization in order to avoid race conditions, i.e. threads overwriting each other's result, thus eliminating the performance improvements that this work tries to achieve. In order to compute $S_p(m)$ and $f_{LS}(m)$, a divide and conquer approach is utilized. Segments of the buffers are assigned to different CUDA threads in order to compute the partial sums and then, the total sum is computed iteratively by one thread. Both sums are computed simultaneously since they are independent from each other. By computing the partial sums, the number of linear operations needed is reduced, thus minimizing the computational cost of the sum calculation. The number of threads N_T used for computing each sum is defined as:

$$N_T = \left\lceil \frac{N(m)}{N_{SE}} \right\rceil \quad (3)$$

with N_{SE} being the number of elements in each segment.

After the computation of $S_p(m)$ and $f_{LS}(m)$, the fitness $F(m)$ is computed and transferred to the main memory of the system. In terms of execution time, this memory transfer costs only the overhead of the transfer operation since the size of the transferred data is extremely small.

Following the aforementioned method, two implementations for the computation of $F(m)$ are proposed. The first implementation (named 'GPU-S') computes $F(m)$ for the chromosomes representing the rows of the grid and then for those representing the columns, while the second implementation (named 'GPU-RC') computes simultaneously $F(m)$ for the chromosomes representing both rows and columns since the two tasks are independent, thus increasing the utilization of the GPU.

For the execution of CUDA kernels, the block size is set to the maximum threads per block supported by the graphics card if the number of threads needed is larger than this number, or to the nearest multiple of 32 if the number of threads needed is less than the maximum threads per block supported. The number of blocks per grid is defined as:

$$BlocksPerGrid = \left\lceil \frac{Threads\ Needed}{ThreadsPerBlock} \right\rceil \quad (4)$$

III. EXPERIMENTAL EVALUATION

In order to evaluate the proposed implementations, their performance was compared with the performance of the CPU implementation used in [16], using the same parameters for the genetic algorithm. The average computation time of $F(m)$ in *ms* for a generation of the genetic algorithm and for both the rows and columns represented by a chromosome is presented on Fig. 3.

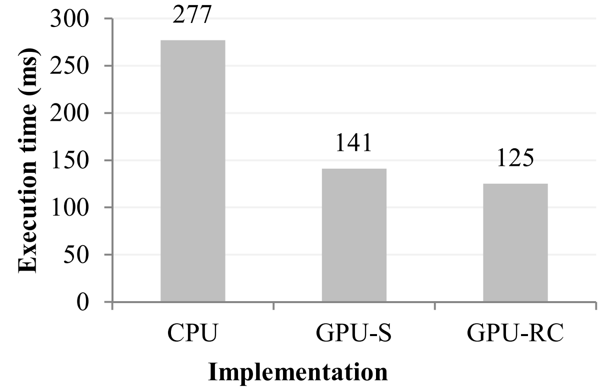

Fig. 3. Average computation time of $F(m)$ for one generation of the genetic algorithm and for both the rows and columns represented by a chromosome.

The GPU-S implementation provides 49% faster performance compared to the CPU implementation, while the GPU-RC implementation provides 55% faster performance.

Each implementation was then tested for several numbers of generations and the average execution time is shown on Fig. 4. The GPU implementations provided faster performance offering an average improvement in speed of 16% for the GPU-S and 20% for the GPU-RC. It is evident that the overall performance improvement is significantly less than the improvement in $F(m)$ computation. This phenomenon can be explained due to the fact that the inevitable memory transfers to and from the GPU dedicated memory at each generation consume a large portion of the execution time.

Performance tests were conducted on a computer equipped with an Intel Core i3 CPU, 4 GB of DDR3 memory and a NVIDIA GeForce GTX 650 Ti graphics card with 1024 MB of GDDR5 memory.

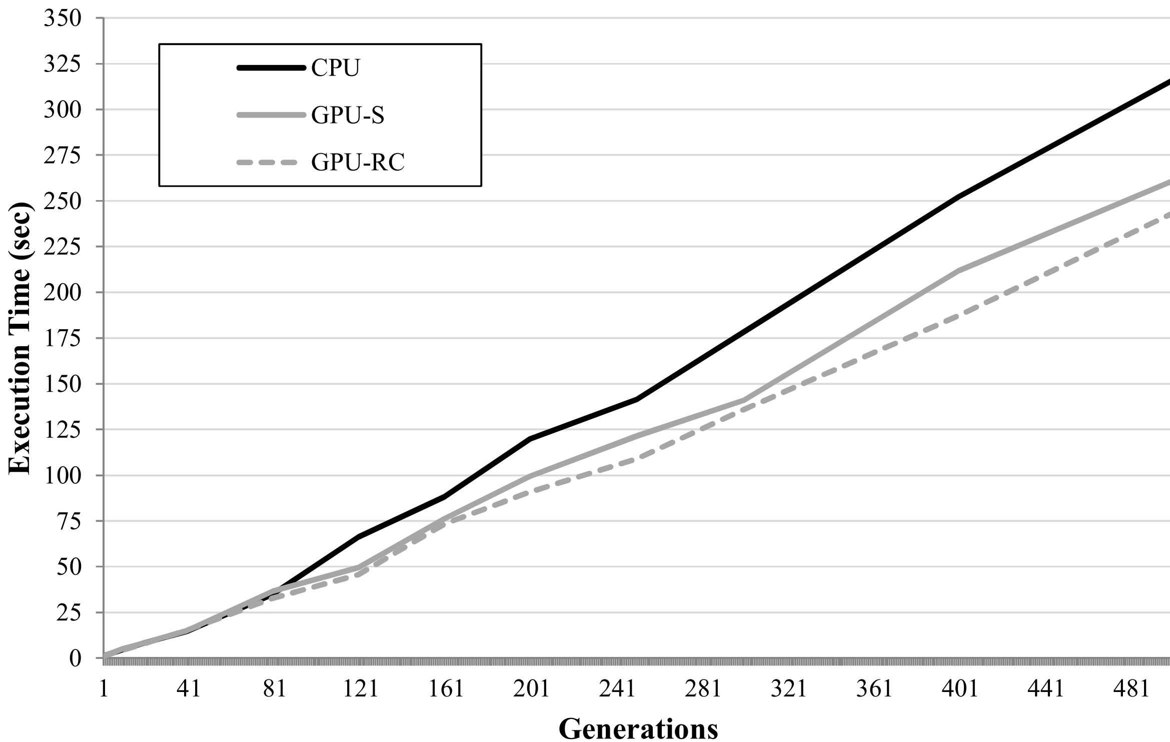

Fig. 4. Average computation time for several generations of the genetic algorithm..

IV. CONCLUSION

In this work, a performance improvement for a microarray image gridding algorithm was presented. Performance is enhanced by harnessing the computational power of the GPU through the CUDA architecture, in order to perform parallel computations. The experimental procedure showed that the proposed method provided up to 55% faster performance for calculating the fitness of a chromosome, and up to 20% overall faster performance for the whole algorithm. Future work could include the calculation of *F(m)* simultaneously for all chromosomes in the population for each generation and also for all the blocks of the microarray image, since both tasks are independent from each other and can be divided to independent steps.

REFERENCES

[1] A. M. Campbell, L. J. Heyer, "Discovering Genomics," in *Proteomics & Bioinformatics*, 2nd ed., Ed. San Francisco: Pearson Benjamin Cummings, 2007, pp. 233-238.

[2] W. B. Chen, C. Zhang, and W. L. Liu, "An Automated Gridding and Segmentation Method for cDNA Microarray Image Analysis," in *2006 Proc. 19th IEEE Symp. Computer-Based Medical Systems*, pp. 893-898.

[3] Y. Tu, G. Stolovitzky, U. Klein, "Quantitative noise analysis for gene expression microarray experiments," in *2002 Proc. National Academy of sciences*, USA, pp.14031-14036.

[4] Biodiscovery Inc. (2005). ImaGene. [Online]. Available: http://www.biodiscovery.com/software/imagene/

[5] M. B. Eisen. (1999). ScanAlyze. [Online]. Available: http://rana.lbl.gov/EisenSoftware.htm

[6] P. Hegde, R. Qi, K. Abernathy, C. Gay, S. Dharap, R. Gaspard et al., "A concise guide to cDNA microarray analysis," *Biotechniques*, vol. 29, no. 3, pp. 548–562, Sept. 2000.

[7] P. Bajcsy, "Gridline: automatic grid alignment in DNA microarray scans," *IEEE Trans. Image Processing*, vol. 13, no. 1, pp. 15-25, Jan. 2004.

[8] P. Bajcsy, "An Overview of DNA Microarray Grid Alignment and Foreground Separation Approaches," EURASIP Journal on Applied Signal Processing, vol. 2006:080163, pp. 1-13, Apr. 2006.

[9] D. Verdnik, "Guide to Microarray Analyses," GenePix Pro: MDS Analytical Technologies, 2004.

[10] Y. Wang, Q.M. Marc, K. Zhang, Y.F. Shih, "A hierarchical refinement algorithm for fully automatic gridding in spotted DNA microarray image processing," *Information Sciences*, vol. 177, no. 4, pp. 1123-1135, Feb. 2007.

[11] J. Angulo, J. Serra, "Automatic analysis of DNA microarray images using mathematical morphology," *Bioinformatics*, vol. 19, no. 5, pp. 553-562, 2003.

[12] N. Deng, H. Duan, "The Automatic Gridding Algorithm based on projection for Microarray Image", in *2004 Proc. Int. Conf. Intelligent Mechatronics and Automation*, pp. 254-257.

[13] G. Antoniol, M. Ceccarelli, "A markov random field approach to microarray image gridding," in *2004 Proc. 17th Int. Conf. Pattern Recognition*, pp. 550-553.

[14] H. Y. Jung, H. G. Cho, "An automatic block and spot indexing with k-nearest neighbors graph for microarray image analysis," *Bioinformatics*, vol. 18, no.1, pp. 141–151, Oct. 2002.

[15] N. D. Lawrence, M. Milo, M. Niranjan, P. Rashbass, S. Soullier, "Reducing the variability in cDNA microarray image processing by Bayesian inference," *Bioinformatics*, vol. 20, no. 4, pp. 518–526, Mar. 2004.

[16] E. Zacharia, D. Maroulis, "An Original Genetic Approach to the Fully-Automatic Gridding of Microarray Images," *IEEE Trans. Medical Imaging*, vol. 27, no.6, pp. 805-813, June 2008.

[17] NVIDIA Corporation, "NVIDIA CUDA programming guide", version 4.2., 2013.

[18] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi", Whitepaper, 2009.

[19] T. Back, Evolutionary algorithms in theory and practice, Ed. Oxford: Oxford University Press, 1996.

[20] D. E. Goldberg, "Genetic Algorithms," in *Search, Optimization & Machine Learning*, Ed. Boston: Addison-Wesley Publishing, 1989.