Compute pairwise Manhattan distance and Pearson correlation coefficient of data points with GPU

Dar-Jen Chang, Ahmed H. Desoky, Ming Ouyang, Eric C. Rouchka

Computer Engineering & Computer Science Department
University of Louisville
Louisville, Kentucky 40292
United States of America

Abstract

Graphics processing units (GPUs) are powerful computational devices tailored towards the needs of the 3-D gaming industry for high-performance, real-time graphics engines. Nvidia Corporation released a new generation of GPUs designed for general-purpose computing in 2006, and it released a GPU programming language called CUDA in 2007. The DNA microarray technology is a high throughput tool for assaying mRNA abundance in cell samples. In data analysis, scientists often apply hierarchical clustering of the genes, where a fundamental operation is to calculate all pairwise distances. If there are n genes, it takes $O(n^2)$ time. In this work, GPUs and the CUDA language are used to calculate pairwise distances. For Manhattan distance, GPU/CUDA achieves a 40 to 90 times speed-up compared to the central processing unit implementation; for Pearson correlation coefficient, the speed-up is 28 to 38 times.

KEY WORDS
Parallel and distributed computation, hierarchical clustering, similarity and dissimilarity metrics

1. Introduction

Graphics processing units (GPUs) on commodity video cards have evolved into powerful computational devices tailored towards the needs of the 3-D gaming industry for high-performance, real-time graphics engines. In the past, software development on GPUs has been geared exclusively towards graphics through the use of languages such as OpenGL shading language and Direct3D high-level shader language. In 2006, Nvidia Corporation released a new generation of GPUs designed for general purpose use. These G80 series GPUs provide up to 128 stream processors and support 12,288 active threads. This new architecture facilitates efficient general purpose computing on GPUs (GPGPUs). In 2007, Nvidia Corporation released an extended C language for GPU programming called CUDA [9], short for Compute Unified Device Architecture. Using CUDA, innovative data-parallel algorithms can be implemented in general computing terms to solve many important and non-graphics applications such as database searching and sorting, medical imaging, protein folding, and fluid dynamics simulation.

In bioinformatics, two kinds of data are being massively accumulated in public databases at ever increasing rates. The first kind of data are biological sequence data, such as gene sequences, protein sequences, and genome sequences. Given novel sequences, researchers routinely conduct sequence similarity searches against these public sequence databases. Even when a linear time heuristic method, such as BLAST [1], is employed, the sheer sizes of the databases entail long waits for the search results. To obtain acceptable response time, researchers have used high-performance computing devices such as large-scale PC clusters to speed up the searches. The second kind of massive data in bioinformatics come from high throughput assaying techniques, such as the DNA microarray technology [2], which assays messenger RNA (mRNA) abundance of cell samples. A DNA microarray simultaneously measures the mRNA levels of thousands to tens of thousands of genes in one experiment. Scientists usually conduct studies consisting of dozens of microarray experiments. The data are collected in a matrix where the rows correspond to the genes, the columns correspond to the experimental conditions, and each entry in the matrix is the abundance of the gene under the particular condition. During the exploratory phase of data analysis, scientists often apply (agglomerative) hierarchical clustering on the genes [4], under the assumption that genes with similar regulatory pathways or with similar biological functions will cluster together.

A common feature of sequence database searching and microarray data clustering is data-parallelism, or single instruction multiple data (SIMD) computation. That is, the same calculation is performed over numerous pieces of computationally independent data. In light of the massively data-parallel capability and wide availability of GPUs, GPGPUs provide a very attractive and cost-effective way to analyze bioinformatics data. Indeed, in recent years there

Corresponding author: Ming Ouyang, ming.ouyang@louisville.edu

have been some studies in the literature that specifically exploit GPGPU in sequence alignment [7], [8], [11]. As for clustering, the k-means method [5] is a favorite among computer scientists, and there are several CUDA implementations of k-means that can be easily located on the world wide web. Unfortunately, it is difficult to delineate the development chronology and authorship because many authors did not pursue formal publication. For the clustering of DNA microarray data, the hierarchical clustering method is the favorite among experimental scientists, mostly due to their early adoption of Michael Eisen's Cluster software [4]. Moreover, time course or dose response studies also favor hierarchical clustering. Eisen's Cluster is very slow when the data matrix is large. Thus, a fast hierarchical clustering tool will facilitate timely discoveries in the life sciences. A fundamental operation in hierarchical clustering is to calculate all pairwise distances. Zhang and Zhang [13] used a shader language and achieved 2-4 times of speed-up compared to a central processing unit (CPU) implementation. Chang *et al.* [3] used GPU and CUDA to compute pairwise Euclidean distance and achieved 20 to 44 times of speed-up to the CPU.

In Section II, CUDA implementations of pairwise Manhattan distance and Pearson correlation coefficient are presented. In particular, Pearson correlation coefficient is the most commonly used similarity metric by experimental scientists. In Section III, GPU/CUDA computational results are presented, which show a 40 to 90 times speed-up for Manhattan distance and a 28 to 38 times speed-up for Pearson correlation coefficient than CPU. In Section IV, possible improvements and future work are discussed.

2. Data and Methods

2.1. Data

In a cDNA microarray experiment [2], a microarray chip is hybridized to the mixture of a control sample and a treated sample. There are thousands to tens of thousands spots on the chip. Each spot typically yields two 16-bit integers; they are the abundance indices of the gene in the control and treated samples. A common practice in microarray data processing is to take the base-2 logarithm of the ratio of treatment over control. Thus a positive number indicates up-regulation of the gene in the treated sample, a negative number indicates down-regulation, and a zero means there is no change. Microarray data are known to be very noisy, despite various preprocessing, normalization, and transformation [10]. Therefore, although many software tools generate double-precision numbers, single-precision (4 bytes, or `float` in C) is more than adequate to store these resulting numbers.

Let n be the number of genes and let m be the number of experimental conditions. In data mining, n is known as the number of objects and m is the number of features. The data used in the present study were retrieved from Stanford Microarray Database [12]. In order to compare the performance of program code on data with different sizes, n takes a value from 4,096, 8,192, and 12,288, and m takes a value from 16, 32, 48, and 64. These numbers represent typical ranges of microarray data clustering. For example, the budding yeast (*Saccharomyces cerevisiae*) genome has 6,300 genes and the human genome has 20,000 genes. In most experimental conditions, the majority of the genes will not be perturbed by the treatments. Thus researchers often select a subset of the genes for cluster analysis.

```
void cpuManhattan(float *out, float *in,
                  int n, int m){
  int i, j, k;
  float dist, tmp;

  for(i=0;i<n;i++){
    out[i*n + i] = 0.0;
    for(j=i+1;j<n;j++){
      dist = 0.0;
      for(k=0;k<m;k++){
        tmp = in[i*m + k] - in[j*m + k];
        dist += ((tmp<0)?-tmp:tmp);
      }
      out[i*n + j] = dist;
    }
  }
}
```

Figure 1. The CPU C code that computes the upper triangle of the matrix of pairwise Manhattan distances

2.2. CPU C code

Let us assume that the data are an $n \times m$ matrix M, stored in a `float` array called `in` in row-major order. The code needs compute an $n \times n$ matrix of Manhattan distances or Pearson correlation coefficients, to be stored in a `float` array called `out` in row-major order. Because both Manhattan distance and Pearson correlation coefficient are symmetric, only the upper triangle of `out` needs to be computed.

Manhattan distance (also known as the L_1 distance) between rows i and j is defined as

$$L_1(i,j) = \sum_{k=1}^{m} |M_{ik} - M_{jk}|. \qquad (1)$$

It is straightforward to compute Manhattan distance. Figure 1 has the CPU C code for pairwise Manhattan distance.

Pearson correlation coefficient (ρ) between rows i and j is defined as

$$\rho(i,j) = \frac{1}{m-1} \sum_{k=1}^{m} \left(\frac{M_{ik} - \text{avg}(i)}{\text{std}(i)} \right) \left(\frac{M_{jk} - \text{avg}(j)}{\text{std}(j)} \right) \qquad (2)$$

```
void cpuRho(float *out, float *in,
            int n, int m){
  int i, j, k;
  float x, y, a1, a2, a3, a4, a5;
  float avgX, avgY, varX, varY, cov, rho;

  for(i=0;i<n;i++){
    out[i*n + i] = 1.0;
    for(j=i+1;j<n;j++){
      a1 = a2 = a3 = a4 = a5 = 0.0;
      for(k=0;k<m;k++){
        x = in[i*m+k];
        y = in[j*m+k];
        a1 += x;
        a2 += y;
        a3 += x*x;
        a4 += y*y;
        a5 += x*y;
      }
      avgX = a1/m;
      avgY = a2/m;
      varX = (a3 - avgX*avgX*m)/(m-1);
      varY = (a4 - avgY*avgY*m)/(m-1);
      cov = (a5 - avgX*avgY*m)/(m-1);
      rho = cov/sqrtf(varX*varY);
      out[i*n + j] = rho;
    }
  }
}
```

Figure 2. The CPU C code that computes the upper triangle of the matrix of pairwise Pearson correlation coefficients

For each pair of i and j, three separate and sequentially executed `for` loops are needed if a straightforward implementation is employed: one loop to calculate avg(i) and avg(j), followed by another loop to calculate std(i) and std(j), and followed by the other loop to calculate Equation (2). However, it is undesirable to employ three separate loops sequentially processing a large array with modern paged memory management, for the loops may incur many page faults. Figure 2 has the CPU C code for pairwise Pearson correlation coefficient using one `for` loop (indexed by `k`) for each pair of i and j. The code in Figure 2 can be further improved for speed by moving the calculation of `a1` and `a3` from the innermost loop to the outermost one so that the computation for avg(i) and std(i) is not repeated for every value of j. However, it is thought that clarity in Figure 2 is more important than the fastest possible execution speed.

2.3. The CUDA language

Nvidia Corporation has released a programming guide to the CUDA language [9], which is an extension of the C language [6]. Briefly, the Single Program Multiple Data (SPMD) code is written in a GPU *kernel* function, the data to be operated on are copied from CPU RAM to the video memory of the device, and the C program running on the CPU initiates the data-parallel computation via a kernel function call. A GPU kernel function contains the code that will be executed simultaneously by the GPU processors, and CUDA uses the function type qualifier `__global__` to declare that a function is a GPU kernel function. For example,

```
__global__ void gpuManhattan(...){
  ...
}
```

tells the CUDA compiler *nvcc* that `gpuManhattan` is to be executed by the GPU.

The CUDA language supports a large number of threads. The threads are organized into *blocks*, where there may be up to 512 (2^9) threads in each block. The blocks are further organized into a *grid* of up to $(2^{16}-1)\times(2^{16}-1)$ blocks. CUDA offers one- and two-dimensional grids and one-, two-, and three-dimensional blocks of threads. For example, let `numRows` be the number of rows of the data matrix, and the size of the matrix of pairwise distances is `numRows x numRows`. The code

```
dim3 block(16,16);
dim3 grid(numRows/16, numRows/16);
```

declares that there are 256 threads in a block (organized as a 16 by 16 square), and the blocks are arranged in a `numRows/16` by `numRows/16` grid. It is assumed that `numRows` is a multiple of 16. If it is not the case, one can add additional entries to the data matrix to make it so, and trim the pairwise distance matrix later on to remove the additional entries.

The GPU device provides *registers* and *local memory* for each thread, a *shared memory* for each block, and a *global memory* for the entire grid of blocks of threads. Although all threads execute the same GPU kernel function, a thread is aware of its own identity through its block and thread indices, and thus a thread can be assigned a specific portion of the data on which it can perform computation. For example, a thread can locate its own position among the blocks and the threads using the following code:

```
int bx = blockIdx.x, by = blockIdx.y;
int tx = threadIdx.x, ty = threadIdx.y;
```

Often times the threads of the same block need to coordinate their computation. As an example, the threads are simultaneously updating a matrix with several stages of computation, and a new stage may only start after the previous stage is completely finished. Thus it is necessary for threads that have already finished one stage to wait for other threads. CUDA provides the function `__syncthreads()` that serves as a barrier. A thread is blocked from further execution if it finishes its work earlier than the rest, and the threads in the same block are released after all threads have reached the barrier.

CUDA provides function calls to allocate memory on the GPU device, and to copy data from CPU RAM to the device memory and vice versa. Typical code looks like the following:

```
cudaMalloc((void**)&gpuInput, inSze);
```

```
cudaMemcpy(gpuInput, input, inSize,
           cudaMemcpyHostToDevice);
```

Then from the C code that is executed by the CPU, one may initiate the execution of the grid of blocks of threads on the GPU as follows:

```
gpuPdist<<<grid, block>>>(gpuOutput,
  gpuInput, otherParameters);
```

After the GPU kernel function exits, the results may be copied from the device memory back to CPU RAM.

The shared memory for a block of threads is fast, yet it is limited in size. One strategy to attain high performance is for the threads in the same block to collaborate on loading data that they all need from the global memory to the shared memory. The shared memory is further partitioned into banks. The threads in the same block may access different banks simultaneously, yet a memory bank conflict will serialize the threads involved in the conflict. Thus another strategy for high performance is to avoid bank conflicts as much as possible.

2.4. GPU CUDA code

In this section, the CUDA algorithms for pairwise Manhattan distance and Pearson correlation coefficient are presented. The data array `in` is $n \times m$ in row-major order of n genes and m microarray experiments, and an $n \times n$ matrix `out` of pairwise (gene-gene) distances will be computed.

Figure 3 illustrates the idea of the CUDA algorithms. Let us assume that a thread needs to calculate the entry indexed by (i, j) in the matrix `out` where i is `16*by` and j is `16*bx`. Row `16*by` of the matrix `in` is first copied to shared memory for fast access. While the copy of row `16*by` is in shared memory, those threads that are responsible for entries $(i, j+1), (i, j+2), \ldots, (i, j+15)$ of the matrix `out` can all use the same copy for fast memory access. Similarly, after row j of the matrix `in` is copied to shared memory, the threads that are responsible for entries $(i, j), (i+1, j), \ldots, (i+15, j)$ can use the same copy. Thus it is advantageous to have square blocks of threads that share the data in the shared memory. Blocks of 16 by 16 threads are used because CUDA limits the maximum number of threads in a block to 512.

Figure 4 has the GPU CUDA code that computes the pairwise Manhattan distances. The algorithm uses one thread for one entry in `out`. Thus there are n^2 threads. The threads are organized into 16×16 two-dimensional blocks, and the blocks are organized into an $n/16 \times n/16$ two-dimensional grid. A thread orients itself through its block and thread indices in the following way:

```
int bx = blockIdx.x, by = blockIdx.y;
int tx = threadIdx.x, ty = threadIdx.y;
```

With the above coordinate system, a thread is responsible to calculate the entry in the matrix `out` at row `by*16 + ty` and column `bx*16 + tx`, and it coordinates its

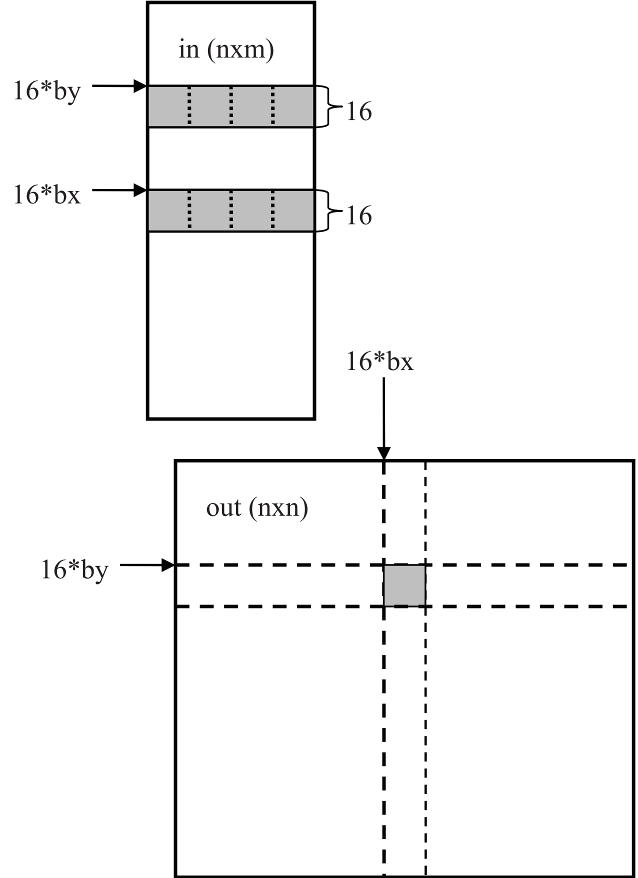

Figure 3. Each block (of 16 by 16 threads) computes one sub-matrix of `out`, and the threads work on one pair of aligned sub-matrices of `in` at a time.

calculation with the other 255 threads in its block in the following manner. During each iteration through the outer `for` loop, the 256 threads first load the two 16 by 16 sub-matrices of `in` anchored by the variables `y` and `x` at the corresponding upper left corners. These sub-matrices are aligned as depicted in Figure 3. After the threads are synchronized, each of them calculates and accumulates its own partial Manhattan distance in the variable `s`. Then the threads need to be synchronized again before proceeding to the next pair of 16 by 16 sub-matrices. A subtle yet very important point in the code is that the array `Xs` in the shared memory is transpose of its image in the matrix `in`. Through the transpose of `Xs`, the number of bank conflicts is reduced in the shared memory access by a factor of 16 and dramatically speed up the calculation.

Figure 5 has the GPU CUDA code that computes the pairwise Pearson correlation coefficients. It is a combination of the algorithm in the CPU C code in Figure 2 and the data loading strategy in the GPU CUDA code in Figure 4.

```
__global__ void
gpuManhattan(float *out, float *in,
             int n, int m){
  __shared__ float Xs[16][16];
  __shared__ float Ys[16][16];
  int bx = blockIdx.x, by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;
  int xBegin = bx * 16 * m;
  int yBegin = by * 16 * m;
  int yEnd = yBegin + m - 1;
  int x, y, k, o;
  float s = 0.0;

  for(y=yBegin,x=xBegin;y<=yEnd;
      y+=16,x+=16){
    Ys[ty][tx] = in[y + ty*m + tx];
    Xs[tx][ty] = in[x + ty*m + tx];
    //*** note the transpose of Xs
    __syncthreads();

    for(k=0;k<16;k++)
      s += fabs(Ys[ty][k] - Xs[k][tx]);
    __syncthreads();
  }
  o = by*16*n + ty*n + bx*16 + tx;
  out[o] = s;
}
```

Figure 4. The GPU CUDA code that computes the pairwise Manhattan distances

3. Results

The performance comparisons are conducted with the following hardware and software setup. An Nvidia Tesla C870 GPU card is installed on a desktop computer with an Intel Pentium D 3GHz CPU and 2GB RAM. The Tesla C870 has a 128-processor core and 1.5GB video memory. The desktop operating system is Microsoft Windows XP Service Pack 3, and the C and CUDA code is compiled by Microsoft Visual Studio 2005, with optimization set for maximum speed (/O2) and floating point model set for "Strict." The computation time is taken by the CUDA timer utility, and the time to copy the input matrix to the GPU card and to copy the output matrix back to the CPU RAM is included in the time for the CUDA code.

As described in Section II, DNA microarray data downloaded from Stanford Microarray Database [12] are used in the computational performance comparisons. The values of n (number of rows) are 4,096, 8,192, and 12,288; the values of m (number of columns) are 16, 32, 48, and 64. Table 1 contains the results. For Manhattan distance, the GPU CUDA code has a 40 to 90 times speed-up than the CPU C code. For Pearson correlation coefficient, the GPU CUDA code has a 28 to 38 times speed-up.

```
__global__ void
gpuRho(float *out, float *in,
       int n, int m){
  __shared__ float Xs[16][16];
  __shared__ float Ys[16][16];
  int bx = blockIdx.x, by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;
  int xBegin = bx * 16 * m;
  int yBegin = by * 16 * m;
  int yEnd = yBegin + m - 1;
  int x, y, k, o;
  float a1, a2, a3, a4, a5;
  float avgX, avgY, varX, varY, cov, rho;

  a1 = a2 = a3 = a4 = a5 = 0.0;
  for(y=yBegin,x=xBegin;y<=yEnd;
      y+=16,x+=16){
    Ys[ty][tx] = in[y + ty*m + tx];
    Xs[tx][ty] = in[x + ty*m + tx];
    //*** note the transpose of Xs
    __syncthreads();

    for(k=0;k<16;k++){
      a1 += Xs[k][tx];
      a2 += Ys[ty][k];
      a3 += Xs[k][tx] * Xs[k][tx];
      a4 += Ys[ty][k] * Ys[ty][k];
      a5 += Xs[k][tx] * Ys[ty][k];
    }
    __syncthreads();
  }
  avgX = a1/m;
  avgY = a2/m;
  varX = (a3-avgX*avgX*m)/(m-1);
  varY = (a4-avgY*avgY*m)/(m-1);
  cov = (a5-avgX*avgY*m)/(m-1);
  rho = cov/sqrtf(varX*varY);
  o = by*16*n + ty*n + bx*16 + tx;
  out[o] = rho;
}
```

Figure 5. The GPU CUDA code that computes the pairwise Pearson correlation coefficients

matrix size		Manhattan distance			Pearson correlation		
n	m	CPU time	CUDA time	speed up	CPU time	CUDA time	speed up
4096	16	2.76	0.07	39.6	2.59	0.09	28.6
8192	16	11.16	0.28	40.5	10.54	0.36	29.3
12288	16	25.12	0.62	40.8	23.74	0.80	29.6
4096	32	5.29	0.08	66.0	3.86	0.12	33.1
8192	32	21.22	0.32	67.1	15.45	0.46	33.4
12288	32	48.02	0.71	67.7	34.88	1.04	33.6
4096	48	7.21	0.09	79.5	5.15	0.14	36.1
8192	48	29.01	0.36	81.2	20.63	0.57	36.4
12288	48	65.47	0.80	81.5	46.51	1.27	36.6
4096	64	8.93	0.10	88.2	6.46	0.17	38.3
8192	64	35.95	0.40	90.0	25.96	0.67	38.7
12288	64	81.12	0.89	90.8	58.47	1.50	38.9

Table 1. Performance comparisons of CPU C code and GPU CUDA code computing pairwise Manhattan distances and Pearson correlation coefficients. Time unit is seconds.

4. Discussion

The current trend in processor design is to pack multiple cores in a single chip. The challenge then is to fully utilize the numerous cores for parallel computation. General purpose computing on graphics processing units (GPGPUs) is emerging as a cost-effective way to boost the computation capability of desktop computers. With an Nvidia Tesla C870 GPU card (at a retail price of $1,000 US dollars or so) and carefully designed code, routine computation can be sped up by a factor of up to 90. In the past, this kind of improvement would require a Beowulf cluster at a much higher cost.

In the present study, the benchmark goes up to 12,288 genes and 64 experimental conditions. These numbers are constrained by the amount of memory (1.5GB) on the GPU card, which puts a limit on the largest matrix that one may use. If 20 to 40 thousand genes are to be clustered, one needs to split the matrix of pairwise distances into pieces so that each piece fits the RAM of the GPU card.

In agglomerative hierarchical clustering, one needs to perform linkage analysis after the matrix of pairwise distances is obtained. There are many different approaches to linkage analysis. As an example, single linkage will join two existing clusters if the shortest distance between data points in these two clusters is the shortest among all pairs of existing clusters. In this case, the distance matrix does not need to be updated after the two clusters are merged. However, with centroid linkage, the newly formed cluster is represented by the centroid of the data points, and thus the distance matrix needs to be updated. There are many other linkage methods. One possible future work is to implement GPU CUDA code for linkage analysis and distance matrix update.

ACKNOWLEDGEMENTS:
Ming Ouyang is partially supported by an Intramural Research Incentive Grant from the Office of Executive Vice President for Research, University of Louisville. Further support for Ming Ouyang and Eric Rouchka is provided by NIH-NCRR grant P20RR16481 and NIH-NIEHS grant P30ES014443. The contents of this manuscript are solely the responsibility of the authors and may not represent the official views of the National Center for Research Resources, the National Institute for Environmental and Health Science, or the National Institutes of Health.

References

[1] Altschul S, Madden T, Schaffer A, Zhang J, Zhang Z, Miller W, Lipman D. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res* 1997, **25**:3389-402.

[2] Brown P, Botstein D. Exploring the new world of the genome with DNA microarrays. *Nat Genet* 1999, **21**:33–7.

[3] Chang D, Jones NA, Li D, Ouyang M, Ragade RK. Compute pairwise Euclidean distances of data points with GPUs. *Proceedings of the IASTED International Symposium on Computational Biology and Bioinformatics* 2008, 278-283.

[4] Eisen MB, Spellman PT, Brown PO, Botstein D. Cluster analysis and display of genome-wide expression patterns. *Proc Natl Acad Sci U S A* 1998, **95**(25):14863–8.

[5] Hartigan JA. *Clustering Algorithms*, Wiley, New York, 1975.

[6] Kernighan BW, Ritchie DM. *The C Programming Language*, Second Edition, Prentice Hall, Inc., 1988.

[7] Liu W, Schmidt B, Voss G, Muller-Wittig W. Streaming algorithms for biological sequence alignment on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 2007, **18**:1270-81.

[8] Manavski SA, Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics* 2008, **9**(Suppl 2):S10.

[9] Nvidia Corporation, *NVIDIA CUDA Programming Guide, Version 2.0* June 7, 2008. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf

[10] Quackenbush J. Microarray data normalization and transformation. *Nat Genet* 2002, **32 Suppl**:496–501.

[11] Schatz MC, Trapnell C, Delcher AL, Varshney A. High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics* 2007 **8**:474.

[12] Sherlock G, Hernandez-Boussard T, Kasarskis A, Binkley G, Matese J, Dwight S, Kaloper M, Weng S, Jin H, Ball C, Eisen M, Spellman P, Brown P, Botstein D, Cherry J. The Stanford Microarray Database. *Nucleic Acids Res* 2001, **29**:152–5.

[13] Zhang Q, Zhang Y. Hierarchical clustering of gene expression profiles with graphics hardware acceleration. *Pattern Recognition Letters* 2006, **27**:676-81.