An Effective and Efficient Approximate Two-Dimensional Dynamic Programming Algorithm for Supporting Advanced Computer Vision Applications

Alfredo Cuzzocrea

DIA Dept., University of Trieste and ICAR-CNR, Italy

Enzo Mumolo

DIA Dept., University of Trieste, Italy

Giorgio Mario Grasso

CSECS Dept., University of Messina, Italy

Gianni Vercelli

DIBRIS Dept., University of Genova, Italy

Abstract

Dynamic programming is a popular optimization technique, developed in the 60's and still widely used today in several fields for its ability to find global optimum. *Dynamic Programming Algorithms* (DPAs) can be developed in many dimension. However, it is known that if the DPA dimension is greater or equal to two, the algorithm is an NP complete problem. In this paper we present an approximation of the fully two-dimensional DPA (2D-DPA) with polynomial complexity. Then, we describe an implementation of the algorithm on a recent parallel device based on CUDA architecture. We show that our parallel implementation presents a speed-up of about 25 with respect to a sequential implementation on an Intel I7 CPU. In particular, our system allows a speed of about ten 2D-DPA executions per second for 85×85 pixels images. Experiments and case studies support our thesis.

Key words: Two-Dimensional Dynamic Programming, CUDA Platform, Computer Vision, Intelligent Systems

Email addresses: alfredo.cuzzocrea@dia.units.it (Alfredo Cuzzocrea),

1 Introduction

In this paper we describe an approximate *Two-Dimensional Dynamic Programming Algorithm* (2D-DPA) running on a CUDA device. *Dynamic programming* (DP), based on the Bellman's Principle of Optimality [1], is a fast, elegant method for finding the global solution to optimization problems. What characterizes a problem suitable for dynamic programming is that solutions to these problems can be formulated as a sequence of simpler problems, and the global optimum is obtained as a sequence of local optima. A classic example may be that of finding the length of a shortest path in a directed graph that has no cycles. Another classical example is that of sequence alignment. Generally-speaking, *computer vision applications* are emerging trends for such a context, and, recently, the research community has devoted a lot of attention to this topic (e.g., [2,3,4,5,6,7,8,9,10,11]).

DP has been applied to various tasks in pattern recognition and computer vision [12,13]. Nowadays, DP is considered a classic optimization method and ever though there are many other optimization techniques available, many researchers still choose DP in their optimization problems because of its conciseness, versatility, and ability to obtain globally optimal solution. Actually, DP is considered an ideal technique for solving a wide variety of discrete optimization problems such as scheduling, string editing, packaging, and inventory management. Of the recent application of DP we can mention tracking [14], stereo [15,16], and elastic image matching [17] problems. Elastic matching is a typical application of 2D-DPA.

DPA was originally developed as a continuous optimization method to obtain the solution efficiently [1]. Angel [18] used analytical DP to smooth interpolated data. Serra and Berthod [19] and Munich and Perona [20] used it for nonlinear alignment of one-dimensional patterns. Recently, Uchida et al. [21] used it in object tracking. DP matching (and its stochastic extension, i.e. *Hidden Markov Models*) is a classical technique for speech recognition [22] and for on-line character recognition [23].

Sequential 1D-DP matching algorithms have been extended to a two-dimensional one by many authors. Truly two-dimensional elastic image matching have been described in [24,25], but the authors have encountered the inherent NP-hardness of the problem [26]. Because of this computational intractability, practical DP-based elastic image matching algorithms employ various approximation strategies, the most popular of which is the limitation of matching flexibility, as the pseudo 2D elastic matching algorithm described in [27]. Another approximation strategy is the partial omission of the mutual

`mumolo@units.it` (Enzo Mumolo), `gmgrasso@unime.it` (Giorgio Mario Grasso), `gianni.vercelli@unige.it` (Gianni Vercelli).

dependency between 4-adjacent pixels (e.g., the tree representation in [28]). Other approximations consist in the introduction of pruning and coarse-to-fine strategies [29], at the cost of global optimality. Notwithstanding these strategies, there is currently no practical DP algorithm that can provide both globally optimal and truly two-dimensional elastic matching. All the conventional DP-based elastic matching algorithms used DP as a combinatorial optimization method. In fact a recent survey [13] reported only combinatorial (i.e., discrete) DP algorithms. Even if the DP optimization problem was originally formulated as a continuous variational problem, it has been discretized and then solved by DP as a combinatorial optimization problem [12].

The paper is organized as follows. Section 2 reports some other CUDA implementations of various DPA based applications. Section 3 describes the Dynamic Programming Algorithms, both in one and two dimensional formulations. It has been shown that the implementation of 2D-DPA has an exponential complexity, therefore in Section 4 we describe an approximation of the two-dimensional algorithm with polynomial complexity. In Section 5, we provide general architecture and funtionalities of the CUDA platform. Section 6 focuses the attention on the CUDA-based implementation of approximate DPA. Section 7 reports experiments showing the benefits that derive from our proposed algorithm. In Section 8, we report a complete case study and related experimental results obtained from the application of the algorithm to *fingerprint verification*. Finally, in Section 9, we report concluding remarks and future work.

2 Related Work

In this Section, we provide an overview of state-of-the-art proposals related to our research. Since the sequential implementation of various types of DPA has high computational demand, many authors implemented the algorithm on Graphics Processing Devices. Two issues have been mainly considered: how to find the best way to parallelize the DPA itself and how to parallelize the problem which has to be solved with DPA.

Many problems have been solved with DPA. The most popular are Stereo Matching in stereo vision, Elastic Matching of images, or various discrete numerical calculus problems. In 2007, a Dynamic Programming-based low density real-time Stereo Matching was implemented on an ATI Radeon $X800$, an early GPU device. They obtained a frame rate from 10 to 20fps [30].

In 2009, Xiao *et al.* address the problem of mapping DPA on *Graphics Processing Units*. They propose a fine-grained parallelization of a single instance of the algorithm that is mapped to the GPU. Steffen *et al.* [31] describe in 2010

an implementation, on a GTX 280, of a numerical framework, called *Algebraic Dynamic Programming*, for encoding a broad range of optimization problems. Depending on the application, they report speedups ranging from about 6 to about 25. In the same year, Congote *et al.* [32] describe the implementation of a *Dense Stereo Matching* algorithm based on Dynamic Programming to recover depth map from two-dimensional images using dynamic programming. They used a number of GPU's available in that year for a parallel implementation of the dynamic programming based algorithm. The sequential implementation was performed with an Intel Pentium processor $E2180$. They found a speed-up of about 16 between the two devices. Stivala *et al.* [33] published in 2010 a paper showing how to parallelize any DPA on a shared memory multi-core computer by means of a shared lock-free hash table, via starting multiple threads that compute the DP recursion in a top-down fashion and memorizing the result in a shared lock-free hash table.

In 2011, Wu *et al.* [34] present the GPU acceleration of an important category of DP problems, called *Non-Serial Polyadic Dynamic Programming*. Since in these problems the parallelism level varies significantly in different stages of computation, they adjusted the thread-level parallelism in mapping a NPDP problem onto the GPU. They report a speedup of about 13 over the previously published GPU algorithm. Nishida *et al.* in 2012 solved an optimization problem with a known dynamic programming solution on a NVIDIA GeForce GTX 580. The problem was the computation of the optimal polygon triangulation of a convex polygon with minimum total weight. The algorithm they published in [35] attained a very high speedup factor of about 250.

3 One- and Two-Dimensional DPA

In this Section, we focus the attention on one- and two-dimensional DPA. A popular way to describe *One-Dimensional Dynamic Programming Algorithms* (1D-DPA) is by means of the *Edit Distance* [36]. The Edit Distance, which finds applications in bio-informatics [37], natural language processing [38] and spoken-word recognition [22], is a way to measure the similarity of two strings or, in other words, to align the two strings. In the following description we extend the Edit algorithm to the comparison of one-dimensional sequences, similar to the comparison between spoken words [22].

Given two one-dimensional sequences, $A = (a_1, a_2, ..., a_i, ..., a_N)$ and $B = (b_1, b_2, ..., ..., b_j, ...b_M)$, the mapping of one sequence to the other is represented by a path M' which starts from cell $(1, 1)$ to cell (N, M). The path is formed by a number of points so that each point k of the path corresponds to a couple of coordinates, $M_k = (i_k, j_k)$. A distance between the two sequences can be defined by the sum of the local distances between the elements of the sequences,

a_i, b_j, computed along a path, namely: $\sum_{k=1}^{|M'|} \|a_{i_k} - b_{j_k}\|$, where $|M'|$ is the length of the path M'. Clearly, there exist a path along which the cumulative distance is minimum. In this case the cumulative distance is the distance between the two sequences:

$$D(A,B) = \frac{\min_{M'} \sum_{k=1}^{|M'|} d(M'_k)}{|M'|} = \\ = \frac{\min_{M'} \sum_{k=1}^{|M'|} d(i_k, j_k)}{|M'|} = \frac{\min_{M'} \sum_{k=1}^{|M'|} \|a_{i_k} - b_{j_k}\|}{|M'|} \tag{1}$$

.

It is worth noting that the factor at the denominator is needed to normalize the distance against different lengths of the optimum path, and it is needed when Equation (1) is used to measure the distance between images.

By Dynamic Programming, the optimization problem of (1) is solved by updating the cumulative distance $D(i,j)$ at each point of the $A-B$ space using the recursion described in Equation (2), which performs the optimal principle of DP.

$$D(i,j) = min \begin{cases} D(i-1,j) + d(i,j) \\ D(i-1,j-1) + 2d(i,j) \\ D(i,j-1) + d(i,j) \end{cases} \tag{2}$$

where $D(1,1) = 2d(1,1)$. The DP recursion described in Equation (2) is represented by in Figure 1.

.

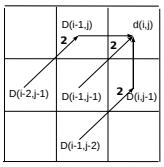

Fig. 1. Graphical representation of DP recursion

After examination of the $A-B$ space, Equation (1) becomes Equation (3):

$$D(A,B) = \frac{D(N,M)}{N+M} \tag{3}$$

where $D(N,M)$ is the cumulative distance at the point (N,M), and the fact that the length of the optimum path is $N+M$ is due to the weight of 2 on the diagonal move. It is important to note that M' which corresponds to the

minimum cumulative distance $D(N, M)$ is the optimal map between A and B and can be used to align one sequence on the other. This operation is called warping. We can think of the goal of matching as bending and stretching the curves to make them identical.

When the sequences to be compared are two-dimensional, ie images, the optimization of Equation (1) can be re-formulated in the following way. Given two images, $X = \{x(i, j)\}$ and $Y = \{y(u, v)\}$, the mapping of one image to the other is represented by the image M''. Each element of M'' corresponds to a couple of pixel's coordinates of the two images, i.e. $M''_{k,l} = (ij_{k,l}, uv_{k,l})$. As in the one-dimensional case, a distance between the two images can be defined as depicted in Equation (4).

$$D(X, Y) = \frac{\min_{M''} \sum_k \sum_l d(M''_{k,l})}{|M''|} =$$

$$= \frac{\min_{M''} \sum_k \sum_l \|x(i, j_{k,l} - y(u, v_{k,l})\|}{|M''|} \tag{4}$$

where, as before, $|M''|$ is a normalization factor.

Similarly to the one-dimensional case, once the distance between the two images is found by solving the optimization described in Equation (4), a warping map M'' is found. Warping one image to the other, can be described as the operation to bend and scratch an image to make it comparable to the other image better than you can.

However, in [26] it has been shown that the optimization described in (4) is NP-complete. As reported before, many authors developed tractable algorithms using various approximation strategies. One early algorithm developed by Levin and Pieraccini in 1992 has a complexity of $O(N^{4N})$ [24]. In [25], a 2D-DPA algorithm is described with a complexity of $O(N^2 9^{2N})$. A continuous and monotonic 2D-DPA algorithm with complexity as $O(N^3 9^N)$ was reported in [39]. In [17] Uchida and Sakoe surveys the elastic matching algorithms proposed so far, seven of which are based on dynamic programming.

The algorithm described in the following Section 4 has a complexity of $O(N^4)$, where M is the images size, assuming that the images have equal height and width.

4 2D Approximate Two-Dimensional DPA

In this Section, we provide the main contribution of our research, i.e. the approximate 2D-DPA.

The algorithm proposed here for the mapping of images is based on the one-dimensional DPA described in Section 3. Consider an image as a vector whose elements are the rows of pixels of the image itself. Let us indicate with $x(i,:)$, $y(i,:)$ the i-th row of pixels of the images X, Y. The X, Y images are thus described as reported in (5).

$$\begin{aligned} X &= [x(1,:), x(2,:), ..., x(i,:), ..., x(N,:)]^T \\ Y &= [y(1,:), y(2,:), ..., y(j,:), ..., y(N,:)]^T \end{aligned} \tag{5}$$

In (5) the images are assumed for simplicity of the same size. The idea of this paper is to apply the one-dimensional DPA algorithm on the two sequences X and Y. We remark that each element of these sequences is an entire row of pixels. The $i-$ th row of X is $x(i,:) = (x_{i,1}, \ldots, x_{i,n}, \ldots, x_{i,N})$ and the $j-$ th row of Y is $y(j,:) = (y_{j,1}, \ldots, y_{j,m}, \ldots, y_{j,N})$. The distance between two elements of X, Y or, in other terms, the distance between two rows of pixels is again performed with one-dimensional DPA. The application of Equation (1) to $x(i,:)$, $y(j,:)$ becomes Equation (6).

$$\begin{aligned} d(x(i,:), y(j,:)) &= \frac{\min_{M'} \sum_{l=1}^{|M'|} d(M'_l)}{|M'|} = \\ &= \frac{\min_{M'} \sum_{l=1}^{2N} \|x_{i,n_l} - y_{j,m_l}\|}{2N} \end{aligned} \tag{6}$$

On the other hand, the application of 1 to X, Y results in Equation (7). In this case the map $\overline{M'}$ is between all the rows of X and Y. As before, $|\overline{M'}|$ is the length of the path of the $\overline{M'}$ map.

$$\begin{aligned} d(X_i, Y_j) &= \frac{\min_{M'} \sum_{l=1}^{|M'|} d(M'_l)}{|M'|} = \\ &= \frac{\min_{M'} \sum_{l=1}^{|M'|} \|x_{i_l,n_l} - y_{j_l,m_l}\|}{2N} \end{aligned} \tag{7}$$

Finally, the distance between the two images is obtained by Equation (8). In

this case the map $\overline{M'}$ is between all the rows of X and Y. As before, $|\overline{M'}|$ is the length of the path.

$$D(X,Y) = \frac{\min_{\overline{M'}} \sum_k d(\overline{M'}_k)}{|\overline{M'}|} =$$

$$= \frac{\min_{\overline{M'}} \sum_k d(X_{i_k}, Yj_k)}{|\overline{M'}|} = \frac{\min_{\overline{M'}} \sum_k \frac{\min_{M'} \sum_{l=1}^{|M'|} d(M'_l)}{2N}}{2N} =$$

$$= \frac{\min_{\overline{M'}} \{ \sum_k \min_{M'} \sum_{l=1}^{|M'|} \|x_{i,n_l} - y_{j,m_l}\| \}}{4N^2} \tag{8}$$

The term at the denominator of Equation (8) is obtained with the following reasoning. Assuming that the images are of equal size of $N \times N$ pixels, the length of the optimum path between the two images is equal to $2N$. The local distances in each point of this path is obtained with other 1D-DPA with paths of length $2N$. The total length is the sum of $2N$ along the $2N$ long path, giving $4N^2$.

5 CUDA Platform: Architecture and Functionalities

In this Section, we provide general architecture and funtionalities of the CUDA platform. First, GPU is a parallel processor initially developed to accelerate graphical applications. In fact, the typical GPU architecture, reported in Figure 2, is a parallel architecture with many computation units, organized in vertex and pixel shader, which are programmable sequences of instructions which respectively allow the transformation from 3D coordinates to 2D and the assignment of color to each pixel of the image. A GPU device is interfaced with a host computer.

In 2006, it was introduced CUDA (*Compute Unified Device Architecture*), designed to overcome many of the obstacles that prevented a smooth non-graphical programming. Instead of dividing the computational resources in vertex and pixel shaders, the CUDA architecture makes use of unified shaders capable of performing any type of shader (vertex, pixel, etc ..). This means that every single ALU on the chip is driven by a program that has as objective to perform general calculations. These ALUs are constructed to comply with the requirements for IEEE arithmetic in single-precision floating point and are designed to use a set of instructions customized to the general calculation rather than one specifically graphic. In addition the executive units have arbitrary read and write access. Furthermore, they can make use of a cache

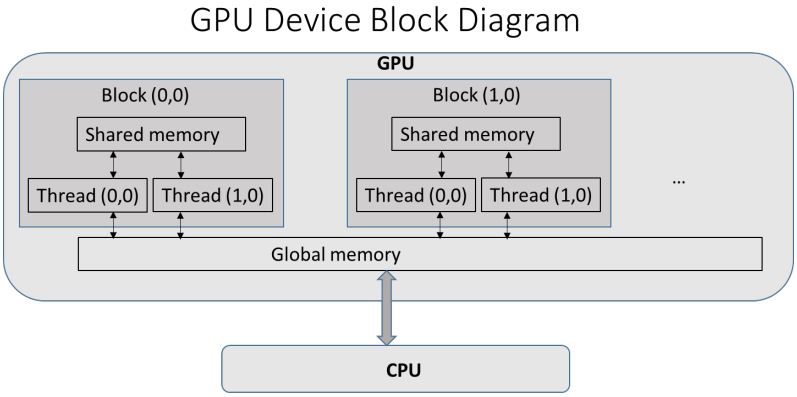

Fig. 2. GPU architecture

maintained in software known as shared memory. All these features have been added to the CUDA architecture to create a GPU suitable to calculate and graph the general-purpose computing.

5.1 CUDA Programming

CUDA programming can be performed using libraries, such as OpenCL, and languages, such as CUDA C/C++, used in this implementation. CUDA C allows the programmer to define C functions (called kernels) that, when called by the CPU (host), are performed on the GPU (device) N times in parallel by N different CUDA threads, and when they end return control back to the host. Threads are organized by CUDA in grids of blocks and scheduled in hardware.

A kernel is defined using the statement `__global__` and returns a `void` parameter. The number of threads running that kernel is specified as the second parameter p_2 inside the brackets $<<< \ p_1, p_2 \ >>>$. Each thread is assigned a unique ID, which is accessible within the kernel code by the variable `threadIdx`. This variable is a three component vector, such that the threads can be identified using an one-dimensional, two-dimensional, three-dimensional index, forming a block of threads with one, two, three-dimensional. This facilitates calculation by elements of domains as vectors, matrices, volumes.

The index of a thread and its ID are related in this way. For a one-dimensional block they are identical. For a two-dimensional block of size (`Dx`, `Dy`), the ID of a thread of index (`x`, `y`), i.e. (`threadIdx.x`, `threadIdy.y`), is (`x + y Dx`). For a three-dimensional block of size (`Dx`, `Dy`, `Dz`), the ID of a thread of index (`x`, `y`, `z`) is (`x + y Dx + z Dx Dy`).

There is a limit to the number of threads per block, since all of the threads

of a block reside on the same core and must share the limited memory of the same.

The blocks, similar to threads, are arranged in one or two-dimensional grids. The number of blocks per grid is specified as the first parameter p_1 inside the brackets $<<< p_1, p_2 >>>$. Each grid block is identified by a one or two-dimensional index accessible within the kernel using the variable `blockIdx`. The block size, ie the number of threads that compose it, is accessible with the variable `blockDim` while the size of the grid blocks is specified by the variable `gridDim` (Figure 3).

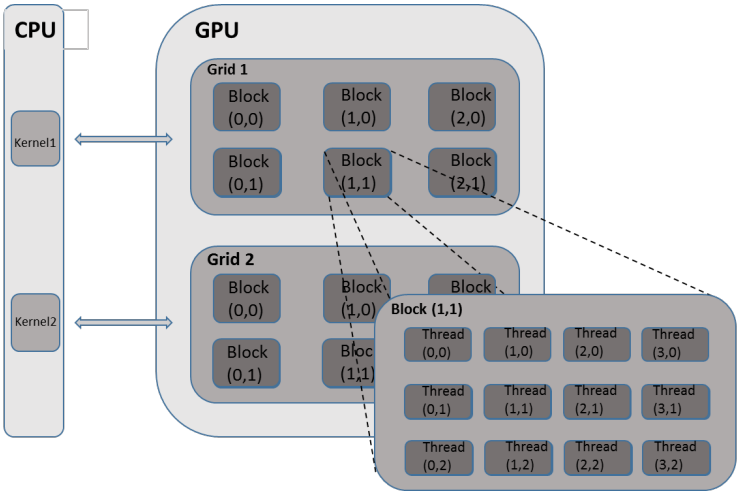

Fig. 3. Grids, blocks and threads in CUDA

There is a maximum number of blocks that can be executed. The blocks must also run independently: it must be possible to execute them in any order, in parallel or in series. This need of independence means that the blocks are scheduled in any order through any number of cores.

Threads within a block can cooperate by sharing data through shared memory and synchronizing their execution to coordinate memory access. Specifically, you can specify synchronization points in the kernel code calling the function `__syncthreads()`; it acts as a barrier to which all threads of the block must wait before each be allowed to continue.

Threads can access data from different memory locations during their execution. Each thread has its own private local memory. Each block has its own shared memory visible to all the threads of the block and with the same time of life of the block. All threads of all blocks have access to a global memory (global memory). There are also two additional spaces of read-only memory accessible by all threads: the constant memory and texture memory. The global, constant, and texture memory survive after different executions of the kernels of the same application.

The memory spaces in the device are typically allocated using `cudaMalloc()` and released with `cudaFree()`; data transfer between host and device is implemented by `cudaMemcpy()`. Since the bandwidth between memory and the host device is much lower than that between two locations device, the programmer should try to minimize data transfers between the host and the device. CUDA also provides functions to allow the use of page-locked host memory (as opposed to traditional pageable host memory allocated with `malloc()`): `cudaHostAlloc()` and `cudaFreeHost()`.

The page-locked buffers have an important property: the operating system ensures that they will reside in physical memory without being stored to disk. The GPU can thus use the direct memory access (DMA) to copy data to and from the host, allowing a significant increase in performance. Both the allocation and de-allocation of memory to create grids of thread blocks are not allowed inside the kernel because they are controlled exclusively by the host.

6 CUDA Implementation of Approximate DPA

This Section focuses the attention on the CUDA-based implementation of approximate DPA. As shown in Equation (7), the computation of the approximate 2D-DPA is obtained by a 1D-DPA where each node of the optimum path is computed by another 1D-DPA. Let us first look at Figure 4.

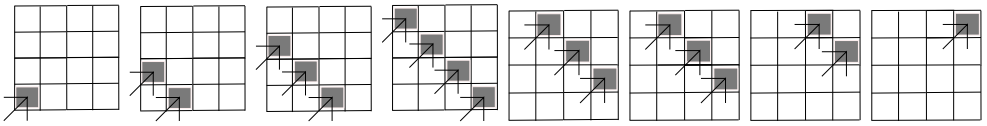

Fig. 4. Explicative graphic sequence of DP operation, clockwise from the left top square. The black cells within a square represent the cells that can be executed in parallel within that square.

It represents various stages of a 1D-DPA between two 1D patterns of length 4 for simplicity. In the left square, top row, we can see the initial 1D-DPA operation. The square drawn in black is the local optimization corresponding to the first parameter of the horizontal pattern and the first parameter of the vertical pattern, as shown in Equation (2) and in Figure 1. We want to compute in parallel most squares as possible. It is evident that, although possible according to the Equation (2) operation, it is not possible to compute in parallel all the first column, as it were possible in a sequential operation: only the square drawn in black can be computed.

After the first black square, i.e. going to the second parameter of the horizontal pattern, the only cells that can be computed in parallel are the two black cells

shown in the second to the left, top row, square of Figure 4. Clearly, the cell at $(1, 1)$ cannot be computed in parallel to the other because of the lack of the values in the bottom and left cells, which must be still computed. Same reasoning at the third parameter of the horizontal pattern, shown in the third to the left, top row, square.

It is clear that the parallel computation of 1D-DPA can be performed in sequences of diagonal cells of increasing size, from 1 to 4. After the diagonal of length 4, shown in the right square of the top row, things change in the sense that the only cells that can be computed in parallel are diagonal sequence of cells, but their number decrease from 4 to 1.

Let us now consider Figure 5, which shows in a simplified way, our mapping of the algorithm on the GPU. It represent, on the vertical axis, a number of patterns, each with 4 elements, from 0 to 3. The horizontal patterns are elaborated one element at a time, from left to right. In the figure it is represented the situation related to the third horizontal element of the horizontal pattern. It is evident that all the cells drawn in black of Figure 5 can be computed in parallel.

In our case, the patterns are the rows of the two images on which Equation (7) is computed. The elements of the patterns are the pixels of the rows. In the experimental Section we assume that the images are of equal size, i.e. $N \times N$, so the images have N rows, and each row have N pixels. In other words, the parallel implementation is implemented taking into account all the templates at once: the relation between the token and the various templates occur simultaneously. This way you do not have a loop that passes through all the templates, nor reset and copies of the flag and the coordinates `x`, a process that would increased the time of execution of the code. Now there is only one cycle `while` which focuses on the number of iterations required to complete the matrix of minima, which is equal to `DimX + DimY - 1`. The variable `DimX` is the width of the matrix, `DimY` the height. As now, all the matrices of the comparisons are considered together, the number of iterations needed to complete the calculation is linked to the size of the larger array.

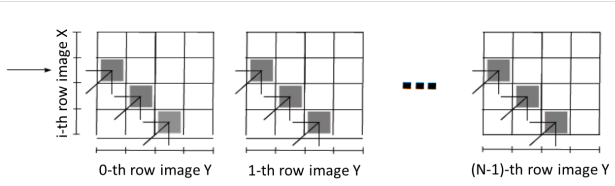

Fig. 5. Simplified representation of the mapping of the algorithm on GPU. Each black cell represent a thread. Horizontal and vertical patterns will be image rows.

The algorithm is as follows:

```
for (r=0;r<N;r++) {
    for (iter=0;iter<=N;iter++) {
```

```
        1D-DPA<<N, iter+1>>>(r,iter, d_cell, ...);
    }
    for (iter=N-1;iter>=1;iter--) {
        1D-DPA<<<N,iter>>>(r,iter, d_cell, ...);
    }
}
```

The outer loop handles all the rows of one image, represented by the variable r. The inner loops perform DP for each pixel of the chosen row. As shown, the first loop perform activate initially one thread per block, then two, then three and so forth, until N. The second loop goes to the last cell, in the right top position of each block.

At this point it is important to remark the following three things.

- One is that initially the images are loaded in the global memory of the GPU, to avoid data transfers from the Host.
- The second is that the operation realized by the code above has the result to compute the local distances matrix between all the rows of the image. On that matrix will be performed another 1D-DPA to solve Equation (7).
- Finally, we remark that, while the distance is obtained by the last cell of the space of the two images, the warping path is obtained by back-tracing from the last cell to the initial. That means that all the information regarding the slopes taken during the DP computation must be stored in data structures and recovered during back-tracing. That means, during the DPAs between the image's rows and during the DPA computed on the local distance matrix.

In fact, the array `d_cell` is virtually a super-matrices that contains all the matrices of individual comparisons. It is basically a three-dimensional matrix in which the linearized coordinates `x` and `y` adds a coordinated `z`, necessary to specify how the token compares template. The main difference in the code of the function `1D-DPA` is therefore to map the indexes of the block with the three-dimensional coordinates of the matrix. Clearly, in addition to storing the backtracking for each association between the lines, you need to store the mapping pixels within the same row.

We now report the pseudo-code of the kernel that performs 1D-DPA on the Kepler GPU. For the sake of simplicity, we denoted as U, V, W the cumulative distances $D(i-1, j-1)$, $D(i, j-1)$ and $D(i-1, j)$ respectively.

```
__global__ void 1D-DPA(int nrtemp, float *d_pdati, float *d_cell, ...)
{
   int z=blockIdx.x/N;
   int y=blockIdx.x \% N;
   int x=d_x[blockIdx.x];
   int i=y + x*N + (d_FrCum[z]*N);
   float dxy, Uval, Vval, Wval;
   char Wfree, Vfree;
   if ((d_checkflag[i]==1) || (y==0 && x==0)) {
```

```
      if ((y==0) && (x==0)) {Uval=0; Vval=1000; Wval=1000; Wfree=0; Vfree=0; }
      else if ((y==0) && (x!=0)) {
            Uval=1000;
            Vval=1000;
            Wval=d_cell[i-N];
            Wfree=d_bWfree[i-N];
            Vfree=0;
      }
      else if ((y!=0) && (x==0)) {
            Uval=1000;
            Vval=d_cell[i-1];
            Wval=1000;
            Wfree=0;
            Vfree=d_bWfree[i-1];
      }
      else {
            Uval=d_cell[i-N-1];
            Vval=d_cell[i-1];
            Wval=d_cell[i-N];
            Wfree=d_bWfree[i-N];
            Vfree=d_bWfree[i-1];
      }
      dxy=d(z,nrtemp,x,y,d_pdati);// distance between pixels
      if ( ( (Wval+dxy) < (Uval+2*dxy) ) && ( Wfree==1 ) )
            if ( ( Wval <= Vval ) || ( bfree==0) ) {
               Vval=Wval+dxy;    // choose W
               d_bWfree[i]=0;
            }
            else {
               Vval=Vval+dxy;    // choose V
               d_bWfree[i]=0;
            }
            else {
               if( ( (Vval+dxy) < (Uval+2*dxy) ) && ( Vfree==1 ) ){
                     Vval=Vval+dxy;    // choose V
                     d_bWfree[i]=0;
               }
               else {
                     Vval=Uval+2*dxy;  // choose U
                     d_bWfree[i]=1;
               }
            }
      d_cell[i]=Vval;
      if (x<d_nftemp[z]-1) {
          d_checkflag[i+1]=1;
          d_checkflag[i+N]=1;
          d_x[blockIdx.x]=d_x[blockIdx.x]+1;
      }
      else if (x==d_nftemp[z]-1 && y<N-1) {
          d_checkflag[i+1]=1;

      }
      else if (x==d_nftemp[z]-1 && y==N-1) {
          d_res[z]=Vval;

      }
   }
}
```

There are a couple of other very important remarks to make at this point.

- The first regards the way the implementation use to synchronize the executions. It has to remember that when a kernel is started, all the created thread share the same code, so how we can stop the thread that cannot

execute like all the cells above the black one in the upper left square of Figure 4 and all the other similar cells in the same figure. We perform synchronization simply with the $d\_checkflag[]$ Boolean variables, which are set when data is available and reset when not.

- The second remark is the use of the flags $Wfree$ and $Vfree$. They are used to impose a path that cannot have two subsequent horizontal or vertical moves, to avoid real paths.

7 Experimental Results

In this Section, we report experiments showing the benefits that derive from our proposed algorithm. The solution of Equation (1) was first obtained using a sequential implementation on a Intel I7 CPU with 8 cores running at 3.07GHz and a memory of 24GB. Then, the algorithm has been rewritten in the CUDA framework and executed on a NVidia Kepler TM GK110 device. This device has many improved features over the previous CUDA Fermi devices, such as dynamic parallelism, the possibility that multiple CPU cores start executions on a single GPU simultaneously, improved Grid management, and enhanced memory subsystem, including additional caching capabilities, more bandwidth , and a faster DRAM I/O implementation. These improvements lead to higher computation capability with respect to the previous Fermi devices.

The algorithm executed on the I7 CPU takes about 2500ms to compute Equation (1) with 85 pixels images. For the same images, the same algorithm running on Kepler TM GK110 takes about 100ms. Of course, the execution times depend on the size of the images. Figure 6 shows the speed-up obtained with difference image sizes.

The algorithm can be applied in many applications as explained previously. In this paper, we considered image matching and image warping applications. We report in this Section only some examples regarding warping. We left image comparison results to future papers. The images were taken from a Uchida's paper and from an image dataset.

In Figure 7, we show three sets of three handwritten characters. The set at the left is written in a vertical manner, the set in the middle is rather inclined and the set at the right is obtained by warping the set in the middle according to the optimum path. Of course the warped image could be passed to an OCR for its recognition. Otherwise, the score obtained with Equation (7) could be used to automatically infer psychological aspects from handwritten writing.

Similar results are shown in Figure 8. In this case the image represent a human face. The warped face (image on the right) could be passed to a face recognition

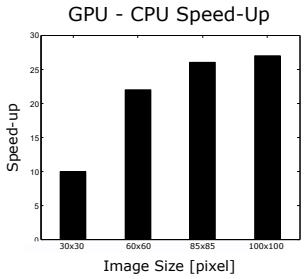

Fig. 6. GPU-CPU execution speedups

Fig. 7. From the left: first handwritten sequence, second handwritten sequence, second image warped onto the first

system. The image at the left and in the middle of this figure are taken from a Uchida paper.

Fig. 8. From the left: first image, second image, second image warped onto the first (Uchida paper)

The human face image shown in Figure 9 is taken from a public dataset. Same considerations as that related to Figure 8 could be done.

Finally, the result shown in Figure 10 are related to a fingerprint image. In this case we can think that the alignment of the fingerprint at the left and that in the middle could simplify the verification process. However, there is a field in fingerprint verification area that address the recovery of orientation of the fingerprint images.

Fig. 9. From the left: first handwritten sequence, second handwritten sequence, second image warped onto the first (public dataset)

Fig. 10. From the left: first fingerprint, second fingerprint, second fingerprint warped onto the first

8 Case Study: 2D Approximate DPA for Supporting Fingerprint Verification

In this Section we report a complete case study and related experimental results obtained from the application of the algorithm to fingerprint verification. The idea is take a given fingerprint as reference and to warp any unknown fingerprint to the reference one. Warping between images is discussed in Section 3. The verification is obtained according to the following rule: if the difference between the reference and the warped fingerprints is below a threshold, then the unknown fingerprint belongs to the same person to whom belongs the reference finger. If the difference is greater than the threshold, then the unknown fingerprint belongs to a different person. This decision rule is motivated from the fact that the warping operation is easier if it is performed between inter-persons fingerprints than between intra-person fingerprints. The first step of our fingerprint verification algorithm is to enhance the raw fingerprint images with the Gabor filters [40]. The second step is to binarize the enhanced fingerprints using a global threshold on the image. The identity of each human user is represented by one Gabor-filtered and binarized fingerprint which is used as a reference for the user to whom the fingerprint belongs. The unknown fingerprints are Gabor-filtered and binarized too. The fundamental additional step is that they are warped to the reference fingerprint with the 2D DPA computed on the Kepler CPU as described in Section 6. The difference between reference and warped fingerprint is evaluated and the total number of pixels of the difference image is evaluated. If this number is greater than a threshold then the unknown fingerprint is verified, otherwise it is not verified. This verification process is quite simple and, if the fingerprints have a size of 100×100 pixels, takes only 100 ms, which is the time to compute the 2D-DPA on the Kepler GPU. A verification example is reported in Figure 11 and

Figure 12, for two different cases. In Figure 11 it is shown, for the first case, from the left, the reference fingerprint of a given user, an unknown fingerprint (but we know it comes from a different imprint of the same user) and the difference from reference and the warped unknown fingerprints. The fingerprint is verified since the number of pixels of the difference image is 20500 and the threshold is set to 27000.

Fig. 11. From the left: reference fingerprint, unknown fingerprint, difference reference-warped unknown (first case)

In Figure 12 it is shown, for the second case, from the left, the reference fingerprint, an unknown fingerprint (but we know it belongs to a different user) and the difference from reference and the warped unknown fingerprint. In this case the fingerprint is not verified since the number of pixels of the difference image is 32200 and the threshold is 27000.

Fig. 12. From the left: reference fingerprint, unknown fingerprint, difference reference-warped unknown (second case)

However, some interruptions in the ridges of the fingerprints may appear in the Gabor-filtered and binarized fingerprints in some noisy fingerprints. The verification results can be improved if the interruptions are restored because the number of valid pixels would be increased.

8.1 Restoring the Fingerprint Ridges Continuity

We have developed a novel technique for restoring the continuity of the fingerprint ridges. The technique is inspired from [41] and takes the following steps:

- thinning of the reference and warped fingerprints;
- reconstruction of the fingerprint ridges.

These two algorithms are summarized in the following sub-sections.

8.1.1 Thinning Algorithm

The Thinning algorithm aims at reducing the ridge thickness, which after Gabor filtering can be several pixels thick, to a only one pixel. In this way the ridges are filiform. Our algorithm is made of the following phases:

- Pre-processing. This operation reduces the ridge thickness to a single medium point.
- Isolated points removal. This is done to reduce noise.
- Discontinuities connection. Small discontinuity zones are searched for and if they are found, they are connected.
- Filling. Pre-processing operation works better if there are no empty regions in the ridges to be thinned. Therefore, white points in the ridges are looked for and, if found, they are filled.
- Removal of not connected pixels. The pixels that have no connections with other pixels are removed using an iterative algorithm based on suitable masks.
- End line correction. When all the previous points are applied, artifacts may be generated at the end of the lines. For correcting these artifacts, an anisotropic operator is applied to modify particular pixel configurations.

8.1.2 Ridge Reconstruction

This phase aims at reconstructing small interruptions that can be still present in the ridge. The ridge reconstruction must be coherent with the contiguous ridges to ensure continuity. For this purpose, a ridge representation should be generated. A representation is obtained by searching the runs of connected pixels. Obviously, a run of connected pixel have a slope. The slopes can be in the range $[-\frac{\pi}{2} \ldots -\frac{\pi}{2}]$. For practical reasons, the slopes are quantized in 13 values, coming from dividing the range of possible slopes in 15^o degrees bands. That is, the first band contains all the slopes from -90^o to -82.5^o, the second band from -82.5^o to -67.5^o and so on until the 13-th band that contains all the slopes from $+82.5^o$ to $+90^o$. First of all, the fingerprint is divided in 4×4 pixels sub-blocks. Each sub-block is labeled with the principal slope of the runs found in the sub-block. This is performed by computing the $R_l(k)$ values, where l is a number of contiguously connected pixels, k is the number of band, and R is the number of runs of length l found in the texel with a slope situated in the k-th band. With the $R_l(k)$ values, a couple of other quantities are computed for each sub-block, as described in Equation 9 and Equation 10.

$$L(k) = \frac{\sum_{l=1}^{n} l^2 R_l(k)}{\sum_{l=1}^{n} R_l(k)} \tag{9}$$

and

$$N(k) = \frac{\sum_{l=1}^{n} R_l^2(k)}{\sum_{l=1}^{n} R_l(k)} \tag{10}$$

These two quantities have the following meaning: Equation (9) measures the presence of long runs and Equation (10) measures the presence of many small runs. Equation (9) and Equation (10) are normalized in the $[0 \dots 1]$ range for each k value over all the sub-block to compare their values. With the normalized values, the predominant slope in each sub-block can be estimated using Equation (11) and Equation (12).

$$P(k) = \max\{0, L(k) - \alpha N(k)\}, \quad k = 1 \dots 13 \tag{11}$$

where $P(k)$ is normalized in the $[0 \dots 1]$ range for each k value in each sub-block. The predominant slope of each sub-block is estimated with Equation (13).

$$P(14) = \min\{N(k)\}, \quad k = 1 \dots 13 \tag{12}$$

$$\text{Predominant Slope} = argmax\{P(k)\}, \quad k = 1 \dots 14 \tag{13}$$

With the predominant slope information, the runs can be finally connected. The connection is performed using a set of simple heuristic rule, reported as follows.

- The connection is performed only for runs belonging to sub-block labeled from 1 to 13.
- In general, the runs are connected according to the minimum distance between the runs.
- Connection between two runs belonging to sub-block of opposite predominant slope is avoided.
- When more than one run is available for connection, only the closes one is connected.

At the end of this Section we report an example of the results that can be obtained from the low level processing techniques described so far, namely the Gabor filtering, the binarization/thinning and the ridge reconstruction. In Figure 13, from left to right, we report an example of a raw fingerprint, of Gabor enhancement, of Thinning and of ridge reconstruction. In the red circle we highlight a case of successful reconstruction.

Fig. 13. From the left: original fingerprint, Gabor-filtered fingerprint, thinned fingerprint, ridge reconstructed fingerprint

8.2 Experimental Results

The algorithm composed by fingerprints enhancement, binarization, and Warping of unknown fingerprints by 2D-DPA implemented on GPU is called the simplified verification algorithm. The algorithm composed by the simplified one plus thinning of the fingerprints plus restoring of the fingerprint ridges is called the overall verification algorithm. Both the algorithms use the verification decision rule based on the computation of the total number of pixels of the difference between reference and warped images. The simplified and overall verification algorithms are tested with the dataset of the second fingerprint verification competition FVC2002 [42]. Generally the performance of a fingerprint verification system are measured in terms of *False Accept Rate* (FAR) and *False Reject Rate* (FRR). FAR is the rate of accepting as verified a fingerprint which in reality belongs to an impostor. FRR is the rate of rejection of valid inputs. Both these measures depend on the threshold of the fingerprint verification algorithm. The threshold value is usually given by the *Equal Error Rate* (EER). ERR corresponds to the threshold which indicates that FAR is equal to FRR.

The FAR and FRR measures and the ERR value evaluated in our test are reported in Figure 14. The dashed red line reports the result of the simplified verification algorithm while the continuous green line is the result of the overall algorithm.

It can be seen that the difference between the simplified and overall algorithms is that the overall one has an ERR less than about 3% of the simplified one.

9 Conclusions and Future Work

In this paper we describe an approximation of two-dimensional dynamic programming. The algorithm has been mapped on a recent GPU device with Kepler architecture. Our results show that two-dimensional dynamic programming can be executed at a rate of about 10 frames per second for images of size 85×85 pixels. Of course other mapping of the algorithm could be devised.

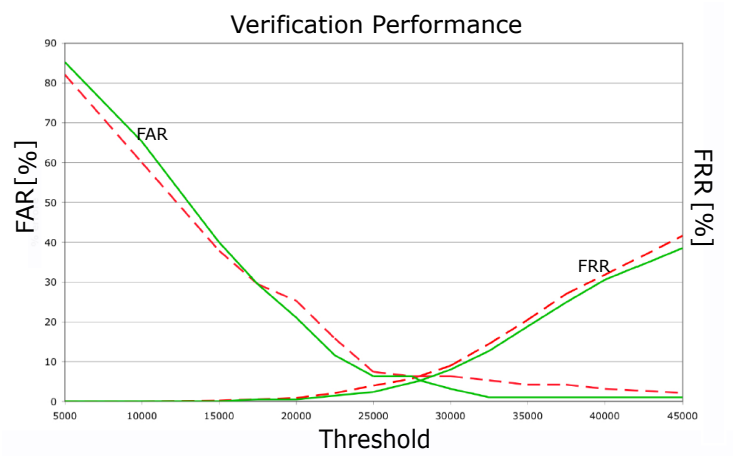

Fig. 14. Performance results of our algorithms on the FVC2002 dataset

In addition to the first experimental campaign, we provided a complete case study and related experimental results obtained from the application of the algorithm to fingerprint verification.

Future work on this topic will be to find better mappings of the algorithm on a GPU on one side, and to apply the algorithm in applications that need real time processing. On the other hand, we plan to extend our framework as to deal with emerging trends of *big data management and analytics* (e.g., [43,44,45,46,47,48,49,50]).

References

[1] R. Bellman, Bynamic Programming, Princeton University Press, 1957.

[2] A. Morales, A. Kumar, M. A. Ferrer, Interdigital palm region for biometric identification, Computer Vision and Image Understanding 142 (2016) 125–133.

[3] Z. Wu, M. Betke, Global optimization for coupled detection and data association in multiple object tracking, Computer Vision and Image Understanding 143 (2016) 25–37.

[4] Y. Long, F. Zhu, L. Shao, Recognising occluded multi-view actions using local nearest neighbour embedding, Computer Vision and Image Understanding 144 (2016) 36–45.

[5] K. Avgerinakis, A. Briassouli, Y. Kompatsiaris, Activity detection using sequential statistical boundary detection (SSBD), Computer Vision and Image Understanding 144 (2016) 46–61.

[6] J. Wang, Z. Xu, Spatio-temporal texture modelling for real-time crowd anomaly detection, Computer Vision and Image Understanding 144 (2016) 177–187.

[7] P. R. Lumertz, L. Ribeiro, L. M. Duarte, User interfaces metamodel based on graphs, J. Vis. Lang. Comput. 32 (2016) 1–34.

[8] K. Zhang, M. A. Orgun, R. Shankaran, D. Zhang, Classifying high dimensional data by interactive visual analysis, J. Vis. Lang. Comput. 33 (2016) 24–36.

[9] W. Wang, M. L. Huang, Q. V. Nguyen, W. Huang, K. Zhang, T. Huang, Enabling decision trend analysis with interactive scatter plot matrices visualization, J. Vis. Lang. Comput. 33 (2016) 13–23.

[10] N. A. M. ElSayed, B. H. Thomas, K. Marriott, J. Piantadosi, R. T. Smith, Situated analytics: Demonstrating immersive analytical tools with augmented reality, J. Vis. Lang. Comput. 36 (2016) 13–23.

[11] C. Lin, W. Huang, W. Liu, S. Tanizar, S. Jhong, Evaluating esthetics for user-sketched layouts of clustered graphs with known clustering information, J. Vis. Lang. Comput. 37 (2016) 1–11.

[12] A. A. Amini, T. E. Weymouth, R. C. Jain, using dynamic programming for solving variational problems in vision, PAMI 12 (9).

[13] P. F. Felzenszwalb, R. Zabih, dynamic programming and graph algorithms in computer vision, PAMI 33 (4).

[14] A. Buchanan, A. Fitzgibbon, interactive feature tracking using k-d trees and dynamic programming, in: Proceedings of CVPR, 2006.

[15] O. Veksler, stereo correspondence by dynamic programming on a tree, in: Proc, of CVPR, 2005.

[16] C. Lei, J. Selzer, Y. H. Yang, region-tree based stereo using dynamic programming optimization, in: Proceedings of CVPR, Vol. 2, 2006, pp. 2378–2385.

[17] S. Uchida, H. Sakoe, survey of elastic matching techniques for hanwritten character recognition, IEICE Transactions Inf. and Sist. (2005) 1781–1790.

[18] E. Angel, dynamic programming for noncausal problems, IEEE Trans. on Automatic Control.

[19] B. Serra, M. Berthod, subpixel contour matching using continuous dynamic programming, in: Proc. of CVPR.

[20] M. E. Munich, P. Perona, continuous dynamic time warping for translation invariant curve alignment with applications to signature verification., in: Proc. of ICCV, 1999.

[21] S. Uchida, I. Fujimura, H. Kawano, Y. Feng, analytical dynamic programming tracker, in: Proc. of ACCV, 2010.

[22] H. Sakoe, S. Chiba, Readings in speech recognition, Morgan Kaufmann Publishers Inc., 1990, Ch. Dynamic programming algorithm optimization for spoken word recognition, pp. 159–165.

[23] C. L. Liu, S. Jaeger, M. Nakagawa, online recognition of chinese characters:the state-of-the-art, PAMI 26 (2).

[24] E. Levin, R. Pieraccini, dynamic planar warping for optical character recognition, in: Proceeding of ICASSP, 1992, pp. 149–152.

[25] Seiichi Uchida, H. Sakoe, a monotonic and continuous two-dimensional warping based on dynamic programming, in: Proc. 14th ICPR, 1998, pp. 521–524.

[26] D. Keysers, W. Unger, elastic image matching is npcomplete, Pattern Recognition Letters 24 (2003) 445–453.

[27] D. Keysers, T. Deselaers, C. Gollan, H. Ney, deformation models for image recognition, IEEE Transactions on Pattern Analysis and Machine Intelligence (2007) 1422–1435.

[28] V. Mottl, S. Dvoenko, A. Kopylov, pattern recognition in interrelated data: the problem, fundamental assumptions, recognition algorithms, in: Proc. of ICPR, 2004.

[29] H. Lester, S. R. Arridge, a survey of hierarchical non-linear medical image registration, Pattern Recognition 32 (1).

[30] Minglun Gong, Y.-H. Yang, real-time stereo matching using orthogonal reliability-based dynamic programming, IEEE TRANSACTIONS ON IMAGE PROCESSING 16 (3) (2007) 879–884.

[31] Parallel Processing and Applied Mathematics, Vol. 6068 of LNCS, Springer, 2010, Ch. GPU Parallelization of Algebraic Dynamic Programming, pp. 290–299.

[32] J. Congote, B. J., I. Barandiaran, R. O., realtime dense stereo matching with dynamic programming in cuda, in: Proc. of CEIG09, Vol. 32, 2009.

[33] Alex Stivala, P. J. Stuckeya, M. G. de la Bandac, M. Hermenegildod, lock-free parallel dynamic programming, Journal of Parallel and Distributed Computing (2010) 839–848.

[34] C. Wu, Y. Ke, H. Lin, W. Feng, optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism, in: Proc. of IEEE 17th International Conference on Parallel and Distributed Systems, 2011, pp. 96–103.

[35] Y. I. Kazufumi Nishida, Koji Nakano, accelerating the dynamic programming for the optimal polygon triangulation on the gpu, in: Algorithms and Architectures for Parallel Processing, 2012, pp. 1–15.

[36] N. Gonzalo, a guided tour to approximate string matching, ACM Computing Surveys (2001) 31–88.

[37] C. Di Neil, P. Pevzner, An Introduction to Bioinformatics Algorithms, MIT Press, 2004.

[38] C. J. Hopfe, Y. Rezgui, E. Mtais, A. Preece, H. Li, Natural Language Processing and Information Systems, LNCS6177, Springer, 2010.

[39] Seiichi Uchida, H. Sakoe, an efficient two-dimensional warping algorithm, IEICE Trans. Inf. and Syst.

[40] Jianwei Yang, L. Liu, T. Jiang, Y. Fan, a modified gabor filter design method for fingerprint image enhancement, Pattern Recognition Letters 24 (12) (2003) 1805–1817.

[41] Enzo Mumolo, spectral domain texture analysis for speech enhancement, Pattern Recognition 35 (10) (2002) 2181–2191.

[42] Dario Maio, D. Maltoni, R. Cappelli, J. L. Wayman, A. K. Jain, FVC2002: second fingerprint verification competition, in: 16th International Conference on Pattern Recognition, ICPR 2002, Quebec, Canada, August 11-15, 2002., 2002, pp. 811–814.

[43] M. Cheung, J. She, Z. Jie, Connection discovery using big data of user-shared images in social media, IEEE Trans. Multimedia 17 (9) (2015) 1417–1428.

[44] M. Chen, S. Mao, Y. Liu, Big data: A survey, MONET 19 (2) (2014) 171–209.

[45] A. Cuzzocrea, L. Bellatreche, I. Song, Data warehousing and OLAP over big data: current challenges and future research directions, in: Proceedings of the sixteenth international workshop on Data warehousing and OLAP, DOLAP 2013, San Francisco, CA, USA, October 28, 2013, 2013, pp. 67–70.

[46] A. Cuzzocrea, F. Furfaro, D. Saccà, Hand-olap: A system for delivering OLAP services on handheld devices, in: 6th International Symposium on Autonomous Decentralized Systems (ISADS 2003), 9-11 April 2003, Pisa, Italy, 2003, pp. 80–87.

[47] A. Cuzzocrea, Analytics over big data: Exploring the convergence of datawarehousing, OLAP and data-intensive cloud infrastructures, in: 37th Annual IEEE Computer Software and Applications Conference, COMPSAC 2013, Kyoto, Japan, July 22-26, 2013, 2013, pp. 481–483.

[48] Byunggu Yu, A. Cuzzocrea, D. H. Jeong, S. Maydebura, on managing very large sensor-network data using bigtable, in: IEEE/ACM CCGrid 2012, Ottawa, Canada, May 13-16, 2012, 2012, pp. 918–922.

[49] S. Felix, A. Csillaghy, A computer vision approach to mining big solar data, in: 2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014, 2014, pp. 27–35.

[50] A. F. Villán, R. Casado, R. Usamentiaga, A real-time big data architecture for glasses detection using computer vision techniques, in: 3rd International Conference on Future Internet of Things and Cloud, FiCloud 2015, Rome, Italy, August 24-26, 2015, 2015, pp. 591–596.