Towards An Effective and Efficient Approximation Algorithm for Advanced Computer Vision Applications based on Two-Dimensional Dynamic Programming

Alfredo Cuzzocrea
DIA Dept., University of Trieste and ICAR-CNR, Italy
alfredo.cuzzocrea@dia.units.it

Enzo Mumolo
DIA Dept., University of Trieste, Italy
mumolo@units.it

Giorgio Mario Grasso
CSECS Dept., University of Messina
Italy
gmgrasso@unime.it

Gianni Vercelli
DIBRIS Dept., University of Genova, Italy
gianni.vercelli@unige.it

Abstract

The Dynamic Programming Algorithm (DPA) was developed in the fifties. However, it is sometimes still used nowadays in various fields because it can easily find the global optimum in certain optimization problems. DPA has been applied to problems of one, two or three dimensions. When the dimension of the problem solved by DPA is equal to one the complexity of the algorithm is polynomial but if the size is greater than one, the complexity becomes NP complete. In such cases a practical implementation of the algorithm is possible only using some approximation. In this paper we present a novel approximation of the two-dimensional Dynamic Programming Algorithm (2D-DPA) with polynomial complexity. We then describe a parallel implementation of the algorithm on a recent Graphics Processing Unit (GPU).

1 Introduction

In this paper we describe an approximation of the two-dimensional Dynamic Programming algorithm (2D-DPA) and its implementation on a GPU device with CUDA architecture. It is well known that the the two-dimensional dynamic programming algorithm is NP-complete. Hence, approximate versions are required for its execution. The approximation we propose here has a polynomial complexity.

The dynamic programming algorithm (DP) is based on the principle of optimality of Richard Bellman [3] and it is an elegant method for finding the global solution to certain optimization problems. Although the optimization problem with dynamic programming was originally formulated as a continuous variational problem, later it was first discretized and then solved as a combinatorial problem (namely discrete) optimization [1]. The application of the dynamic programming algorithm requires that an optimization problem is formulated as a series of simpler problems. The global optimum is obtained by the sequence of local optima. A classic example is to align two sequences of symbols, which has found application in speech recognition [21] and bioinformatics [30].

DP has been applied to various problems of pattern recognition and computer vision [1, 9]. Although there are many other optimization techniques, DP is an ideal technique to solve many discrete optimization problems such as inventory management, scheduling of activities or packaging of objects. Recently Buchanan and Fitzgibbon [4] describe an algorithm that processes multiple hypotheses for tracking mobile objects with dynamic programming. In [31, 15] methods for computing the disparity map of stereo images with dynamic programming are described. Furthermore, the elastic comparison between images as described for example by Uchida [29] is a typical application of the two-dimensional dynamic programming.

While DPA was originally used as a method to efficiently solve optimization problems, [3], Angel [2] uses the analytical DP to smooth iterpolated data. Serra and Berthod [22] and Munich and Perona [19] use the DP algorithm for aligning non-linear one-dimensional patterns. Most recently, Uchida *et al.* [25] use DPA for tracking objects.

The algorithm of DP (and its extension, the stochastic programming, namely the Hidden Markov Models) is a classic technique for recognizing the spoken voice [21] and for the recognition of printed characters [17].

DOI reference number: 10.18293/DMS2016-031

Many researchers have extended one-dimensional dynamic programming algorithms (1D-DP) to the two-dimensional case. The comparison of images by two-dimensional DPA has been described for example in [16, 26], but the authors have met the computational difficulties due to the NP completeness of the problem [14].

To address this computational difficulty, different approximation strategies have been proposed, with the aim of arriving at a solution with polynomial complexity but with a lower level of optimization. The approximation type described in [13] limits the flexibility of the correspondences. Another approximation strategy consists in the partial omission of the mutual dependence between four adjacent pixels, such as the tree representation described in [18].

All elastic comparison algorithms between patterns are based on dynamic programming as a method of combinatorial optimization. A recent overview by Felzenszwalb *et al.* [10] emphasizes that the use of dynamic programming as a discrete optimization method is versatile, and arises in very different low-level and high-level vision problems.

This paper is organized as follows. In Section 2 various applications based on DPA implemented on CUDA architecture are described. Section 3 describes dynamic programming algorithms in one and in a two dimensions. In Section 4 we describe a two-dimensional DPA approximation algorithm with polynomial complexity. This as an improvement of well-known argumentation that state that 2D-DPA has an exponential complexity. Section 5 reports the CUDA-based implementation of our proposed approximate 2D-DPA algorithm. Finally, in Section 6, we report concluding remarks and hints on future work. An extended version of this work appears in [7].

2 Related Work

The two-dimensional dynamic programming algorithm requires a large number of computations. For this reason many authors have implemented the algorithm in parallel form on Graphics Processing Devices (GPU). The main problem that have been considered is how to find the best way to parallelize the 2D-DPA.

Many problems have been solved with the 2D-DPA algorithm. For example, the problem of finding the disparity between stereo images, the problem to compute a distance between images using the so-called elastic matching method, or several discrete numerical problems have been addressed using the two-dimensional dynamic programming algorithm. In 2007, a dynamic programming algorithm to solve the problem of finding the disparity between stereo images has been implemented on an ATI Radeon X800 GPU, one of the first GPU devices [11]. In 2009, Xiao *et al.* [32] address the problem of how to map the dynamic programming on a graphics processing unit. They propose a fine-grained parallelization of a single instance of the DP algorithm that is mapped to the GPU. In the same year Congote *et al.* [6] compute the map of depth from stereo images for three-dimensional display on a GPU using dynamic programming. They use an NVIDIA GPU commercially available in that year, in particular the model GTX 295, achieving an execution speed of 25 frames per second. The heart of their work was the use of optimized algorithms for a GPU execution with stable depth maps and little noise. In 2010, Steffen *et al.*, [23], describe their change to the numerical software environment called Algebraic Dynamic Programming consisting of different optimization problems solved with dynamic programming. Their change has been the implementation of the dynamic programming based algorithms on a GTX 280 GPU. The authors report that the speed-ups range, depending on the application, from about 6 to about 25. Stivala *et al.* [24] published an article in 2010 that shows how to parallelize any DPA on a multicore computer with shared memory by means of a hash table without using primitives for mutual exclusion lock. The authors adopted this strategy: initially they create multiple threads that compute top-down DP recursion storing the result in a hash table with no spin-lock.

3 One- and Two-Dimensional DPA

Dynamic Programming is often explained using the *edit* distance, which measures the number of insertions, deletions and substitutions required for matching two sequences of symbols [20]. In other words the edit distance is a way to measure the similarity of two strings or to align two strings. The *edit* distance has many applications, for example in bioinformatics [8] or in natural language [12]. The *edit* distance algorithm is described as follows. 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 path M' from the cell $(1, 1)$ to the cell (N, M) of the $A - B$ space that gives the minimum accumulated distance represents the mapping of one sequence relative to the other. The path is formed by a series of points in such a way that each point k of the path corresponds to a pair of coordinates of the two sequences, $M'_k = (i_k, j_k)$. The distance between the two sequences is the sum of the local distances $d(M'_k)$ between the two addressed elements of the sequences, a_{i_k}, b_{j_k}, computed along the path M', or $\sum_{k=1}^{|M'|} \|a_{i_k} - b_{j_k}\|$, where $|M'|$ is the length of the path M', namely the number of points k. The distance $\|.\|$ depends on the problem under examination.

The mapping between the two sequences is the path along which the final accumulated distance is minimum. Finally, the distance between the two sequences is the mini-

mum accumulated distance:

$$D(A,B) = \frac{\min_{M'} \sum_{k=1}^{|M'|} d(M'_k)}{|M'|} =$$

$$= \frac{\min_{M'} \sum_{k=1}^{|M'|} d(i_k, j_k)}{|M'|} = \frac{\min_{M'} \sum_{k=1}^{|M'|} \|a_{i_k} - b_{j_k}\|}{|M'|} \quad (1)$$

We note that the factor at the denominator is used to normalize the distance in respect of the length of the optimal path, which may be different. The optimization problem described in (1) is solved by the dynamic programming algorithm. In each point of the $A - B$ space the accumulated cost distance $D(i,j)$ is updated using the recursion shown in (2).

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

where $D(0,0) = 2d(0,0)$. The recursion (2) expresses the optimality principle of dynamic programming. This type of recursion is the *symmetric* form of dynamic programming. It is important to remark that using the recursion (2) the length of the optimal path length just mentioned above, which is the normalization factor, is the sum of the lengths of the sequences A and B, namely $N + M$. The recursion reported in (2) is graphically shown in Fig. 1.

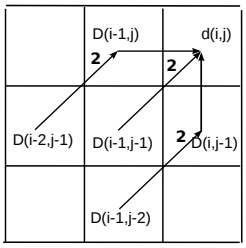

Figure 1. Graphical Representation of DP Recursion

The recursion described in (2) can be implemented according to the following pseudocode, where we call U, V, W, the accumulated cost distances computed at the points, respectively, $D(i-1,j-1)$, $D(i,j-1)$ and $D(i-1,j)$.

```
for i = 0 to N
  for j = 0 to M
  {
   Wval=score[j]; Wfree=free[j];
   if( (Wval+dij)<(Uval+2dij)) && Wfree)
     if(Wval<=Vval)||(!free) {
       Vval=Wval+dij;/* W */
       Vfree=0;
     }
     else {
       Vval=Vval+dij; /* V */
       Vfree=0;
     }
     else {
       if(Vval+dij)<(Uval+2dij) && Vfree)
       {
         Vval = Vval + dij;/* V */
         Vfree = 0;
       }
       else {
         Vval=Uval+2dij;/* U */
         Vfree = 1;
       }
       score[j]=Uval; free[j]=Ufree;
       Uval=val;
  }
 final distance = score[M]/(N + M);
```

As shown in the above pseudocode, after the recursion (2) is applied to all the (i,j) points of the $A - B$ space, equation (1) becomes:

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

where $D(N,M)$ is the final distance between the two one-dimensional sequences, namely the distance accumulated to the point (N,M). The sum $N + M$ at the denominator is the normalization factor.

It is important to remark that the constraints shown in Fig.1 and represented by the *Ufree*, *Vfree* and *Wfree* variables of the above pseudocode cause that the optimum mapping path can not have two consecutive horizontal or vertical movements. Thus, as stated in [21], unrealistic mapping paths are avoided.

Since the path M' is the optimal mapping between the two sequences, it can be used to align one sequence onto the other, which can be performed by inserting or removing a point of one sequence according to the horizontal or vertical moves. This operation is called warping. The goal of the warping is to stretch or shrink one sequence to make it identical to the other.

We now extend the derivation of a distance between two one-dimensional sequences given above to the derivation of a distance between two two-dimensional sequences, namely the images, $X = \{x(i,j)\}$ and $Y = \{y(u,v)\}$. For that, we introduce a mapping plane M", $M" = \{m(k,l)\}$, which maps the two images from the first row of the two images to the last row of the two images. Each of the elements of M" correspond to a pair of coordinates of pixels of the two images, namely $M"_{k,l} = m(k,l) = ((i,j)_{k,l}, (u,v)_{k,l})$. Similarly to the one-dimensional case, a distance between the two images can be defined as reported in (4).

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

$$= \frac{\min_{M"} \sum_k \sum_l \|x(i,j)_{k,l} - y(u,v)_{k,l}\|}{|M"|} \quad (4)$$

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

The plan $M"$ that solves equation (4) is a mapping between the two images. Using the mapping $M"$ one can stretch and shrink an image for overlapping it onto the other one. Similarly to the one-dimensional case this operation is called warping between images.

However, it has been shown that the optimization described in (4) is NP-complete. As previously mentioned, many authors have developed 2D-DPA algorithms using various approximation strategies to make them computationally tractable. The algorithm developed by Levin and Pieraccini in 1992 has a complexity of $O(N^{4N})$, assuming that N is the height and width of the images [16]. Uchida *et.al* described in [27] and in [28] a two-dimensional dynamic programming algorithm with $O(N^3 9^N)$ complexity. Uchida and Sakoe described in [29] various elastic matching algorithms proposed so far, seven of which are based on dynamic programming.

In the following section we describe an algorithm with a complexity of $O(N^4)$.

4 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 (1) to $x(i,:)$, $y(j,:)$ becomes (6).

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

On the other hand, the application of 1 to X, Y results in (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.

$$D(X,Y) = \frac{\min_{\overline{M'}} \sum_k d(\overline{M'}_k)}{|\overline{M'}|} = \frac{\min_{\overline{M'}} \sum_k d(x(i,:), y(j,:))}{|\overline{M'}|} \tag{7}$$

Substituting (6) in (7) we obtain the final expression reported in (8).

$$\begin{aligned} D(X,Y) &= \frac{\min_{\overline{M'}} \sum_k \frac{\min_{M'} \sum_l d(M'_l)}{2N}}{2N} = \\ &= \frac{\min_{\overline{M'}} \{\sum_{k=1}^{2N} \min_{M'} \sum_{l=1}^{2N} \|x_{i,n_l} - y_{j,m_l}\|}{4N^2} \end{aligned} \tag{8}$$

Clearly, the two min operators represent the fact that for computing the optimum path between images with DPA, other optimum paths between the rows are computed with DPA. Recall that we assumed the images are $N \times N$ pixels, then the length of the optimal path between the two images is $2N$. Local distances at any point in this process are obtained with other 1D-DPA with paths length equal to $2N$. The total length is the sum of $2N$ along the long path $2N$, giving the term $4N^2$ at the denominator of (8).

The algorithm is approximated since it is not guaranteed that the warping function is continuous. In fact, the one-dimensional DPA aligns the rows independently of each other. This means that the algorithm can be successfully applied if the difference between the two images is somehow limited.

5 CUDA-based Implementation of Approximate 2D-DPA

Our approximate realization of 2D-DPA is the following: we compute a 1D-DPA between each row of an image and each row of the other image. A matrix of distances between rows is then computed. On this matrix another 1D-DPA is applied. This process is described by equation (7). In Fig. 2 we show how the distance matrix is computed: between i-th row of the first image and the j-th row of the second image a 1D-DPA is computed to fill the (i,j) matrix element.

Let us look at Fig.3. It represents a matrix of threads. The elements $x_{i,0}, \ldots x_{i,3}$ is the $i-$th row of the X image, assuming that its length is 4, and the elements $y_{i,0}, \ldots, y_{i,3}$ is the $j-$th row of the Y image.

Each cell of the matrix computes the DPA recursion. Thus, the accumulated distance at cell $(3,3)$ is the distance between the two rows. In Fig.3 we indicate with arrows the

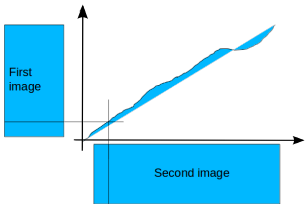

Figure 2. Approximate Mapping Between Two Images

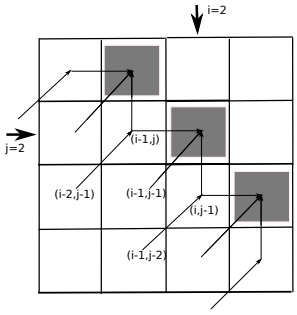

Figure 3. A Matrix of Threads

element (2,2). The corresponding element is indicated with a black square. it is obvious from the DP recursion that the computation of the accumulated distance $D(2,2)$ can be performed only if the distances in the elements whose coordinates are marked between round brackets are already available. This means that the only possible sequence of elements that can be computed in parallel, i.e. simultaneously, is the sequence of elements drawn in black in Fig.3. The parallel computation of elements to the final (M,N) point thus proceeds according to diagonal sequences of elements as depicted in this figure.

Fig. 4, shows how the algorithm can be mapped on the GPU. In this figure, on the vertical axis, a series of image rows, each with 4 elements, are reported. The horizontal models are processed one at a time, from left to right. The figure shows the situation relating to the third horizontal element. The figure shows that all cells drawn black in the Fig. 4 can be computed simultaneously.

From the above it is evident that the parallel implementation is implemented taking account of all models at once. This is accomplished with a single `while` cycle exploring the number of iterations required to complete the matrix minimum, whose size is equal to `DimX` + `dimy` − 1, where `DimX` and `dimy` are the width and height of the matrix.

Since now all of the comparisons matrices are considered simultaneously, the number of iterations needed to complete the calculation is the size of the larger matrix.

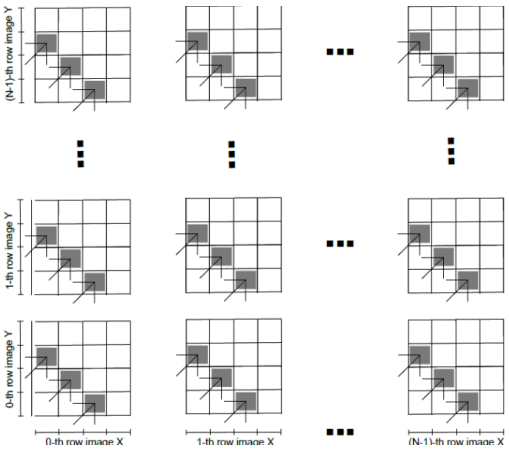

Figure 4. Simplified Representation of the Mapping af the Algorithm on GPU – each black cell represents a thread; horizontal and vertical patterns will be image rows

The following pseudocode shows how we call the kernel.

```
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 rows of an image are indicated by the variable r, which is reported in the outer loop. The one-dimensional dynamic programming used to compute the distance from one row to the other is carried out in the inner loops. The first inner loop creates a thread for each block, the second creates two threads, the third creates three threads and so on, up to N.

At this point it is important to point out three things.

- first, to prevent the transfers of images from the Host, which would slow down the calculation, all the images are initially loaded in the global memory of the GPU
- second, the operation realized by the above pseudo code computes the matrix of local distances between all the rows of the two images. These distances are used in a final dynamic programming to calculate the distance between the two images as shown by (7).
- the third observation is that if we want to perform the warping between the two images, in addition to the

computation of the distance between the two images, we have to recover the mapping plane between the two images. This is achieved by backtracing last initial cell. This means that when calculating the dynamic programming all the information about the algorithm choices must be stored in appropriate data structures. These data structures are used during the backtracing.

In fact, the matrix `d_cell` is a super-matrix that contains all the matrices of the individual comparisons. The matrix `d _cell` is a three dimensional matrix with indices `x`, `y`, `z` necessary to specify how to make comparisons. The main difference in the code of the function `1D-DPA` is the mapping of the block indexes in the three-dimensional coordinates of the matrix. Of course in addition to storing the decisions taken by the algorithm when processing the distance matrix we must also store those taken during the confrontation between the lines.

In the following we report the pseudocode of the kernel that runs the `1D-DPA` algorithm between the lines of the two matrices on the GPU.

```
__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 other important points to consider.

- Recall that when a kernel is started, all the threads execute the same code. However not all of the threads can execute simultaneously, but only those for which the input data have been already computed. So we have to solve this kind of synchronization. The solution that was adopted in this work is simply to use Boolean variables, in particular the variables $d_checkflag[]$, which are set equal to true when the data are available, and false otherwise.

- The second observation is related to the use of the flag $Wfree$ and $Vfree$. These flags are used to impose a path that can not have two consecutive horizontal or vertical movements. These paths are in fact considered incorrect.

6 Conclusions and Future Work

In this paper we describe an approximation of the two-dimensional dynamic programming algorithm in order to make it computationally feasible. The algorithm has been mapped on a recent GPU device. A first example of application of the algorithm can be the automatic analysis of the inclination of handwritten characters. Finding the degree of inclination could be used to infer the physical or mental state of the writer. This work will continue to develop other types of mapping the 2D-DPA on the CUDA architectures in order to reduce the execution time. On the other hand, another research direction consists in enriching the proposed framework with innovative features such as *adaptiveness* (e.g., [5]) and support for *big data processing* (e.g., [33]).

References

[1] A. A. Amini, T. E. Weymouth, and R. C. Jain. Using dynamic programming for solving variational problems in vision. *PAMI*, 12(9), 1990.

[2] E. Angel. Dynamic programming for noncausal problems. *IEEE Trans. on Automatic Control*, 1981.

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

[4] A. Buchanan and A. W. Fitzgibbon. Interactive feature tracking using K-D trees and dynamic programming. In *2006 IEEE (CVPR 2006), 17-22 June 2006, New York, NY, USA*, pages 626–633, 2006.

[5] M. Cannataro, A. Cuzzocrea, and A. Pugliese. XAHM: an adaptive hypermedia model based on XML. In *Proceedings of ACM SEKE 2002, Ischia, Italy, July 15-19, 2002*, pages 627–634, 2002.

[6] J. Congote, J. Barandiarán, I. Barandiaran, and O. E. Ruiz. Realtime dense stereo matching with dynamic programming in CUDA. In *XIX Spanish Computer Graphics Conference, CEIG 2009*, pages 231–234, 2009.

[7] A. Cuzzocrea, E. Mumolo, D. Pirrò, and G. Vercelli. An efficient cuda-based approximate two-dimensional dynamic programming algorithm for advanced computer vision applications. In *IEEE SMC 2016, Budapest, Hungary, October 9-12, 2016*, 2016.

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

[9] P. F. Felzenszwalb and R. Zabih. Dynamic programming and graph algorithms in computer vision. *PAMI*, 33(4), 2011.

[10] P. F. Felzenszwalb and R. Zabih. Dynamic programming and graph algorithms in computer vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(4):721–740, 2011.

[11] M. Gong and Y.-H. Yang. Real-time stereo matching using orthogonal reliability-based dynamic programming. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 16(3):879–884, 2007.

[12] C. J. Hopfe, Y. Rezgui, E. Métais, A. D. Preece, and H. Li, editors. *15th International Conference NLDB 2010, Cardiff, UK, June 23-25, 2010. Proceedings*, LNCS 6177. Springer, 2010.

[13] D. Keysers, T. Deselaers, C. Gollan, and H. Ney. Deformation models for image recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(8):1422–1435, 2007.

[14] D. Keysers and W. Unger. Elastic image matching is np-complete. *Pattern Recognition Letters*, 24(1-3):445–453, 2003.

[15] C. Lei, J. M. Selzer, and Y. Yang. Region-tree based stereo using dynamic programming optimization. In *2006 IEEE (CVPR 2006), 17-22 June 2006, New York, NY, USA*, pages 2378–2385, 2006.

[16] E. Levin and R. Pieraccini. Dynamic planar warping for optical character recognition. In *Proceeding of ICASSP*, pages 149–152, 1992.

[17] C. Liu, S. Jäger, and M. Nakagawa. Online recognition of chinese characters: The state-of-the-art. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(2):198–213, 2004.

[18] V. Mottl, S. Dvoenko, and A. Kopylov. Pattern recognition in interrelated data: The problem, fundamental assumptions, recognition algorithms. In *17th International Conference ICPR 2004, Cambridge, UK, August 23-26, 2004.*, pages 188–191, 2004.

[19] M. E. Munich and P. Perona. Continuous dynamic time warping for translation-invariant curve alignment with applications to signature verification. In *ICCV*, pages 108–115, 1999.

[20] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[21] H. Sakoe and S. Chiba. *Readings in speech recognition*, chapter Dynamic programming algorithm optimization for spoken word recognition, pages 159–165. Morgan Kaufmann Publishers Inc., 1990.

[22] B. Serra and M. Berthod. Subpixel contour matching using continuous dynamic programming. In *CVPR 1994, 21-23 June, 1994, Seattle, WA, USA*, pages 202–207, 1994.

[23] P. Steffen, R. Giegerich, and M. Giraud. *GPU Parallelization of Algebraic Dynamic Programming*, pages 290–299. Springer Berlin Heidelberg, 2010.

[24] A. Stivala, P. J. Stuckey, M. G. de la Banda, M. V. Hermenegildo, and A. Wirth. Lock-free parallel dynamic programming. *J. Parallel Distrib. Comput.*, 70(8):839–848, 2010.

[25] S. Uchida, I. Fujimura, H. Kawano, and Y. Feng. Analytical dynamic programming tracker. In *Computer Vision - ACCV 2010, Queenstown, New Zealand, November 8-12, 2010*, pages 296–309, 2010.

[26] S. Uchida and H. Sakoe. A monotonic and continuous two-dimensional warping based on dynamic programming. In *ICPR 1998, Brisbane, Australia, 16-20 August, 1998*, pages 521–524, 1998.

[27] S. Uchida and H. Sakoe. A monotonic and continuous two-dimensional warping based on dynamic programming. In *Proc. 14th ICPR*, pages 521–524, 1998.

[28] S. Uchida and H. Sakoe. An efficient two-dimensional warping algorithm. *IEICE Trans. Inf. and Syst.*, 1999.

[29] S. Uchida and H. Sakoe. Survey of elastic matching techniques for hanwritten character recognition. *IEICE Transactions Inf. and Sist.*, pages 1781–1790, 2005.

[30] A. Vajdi, N. Haspel, and H. Banaee. A new DP algorithm for comparing gene expression data using geometric similarity. In *IEEE BIBM 2015, Washington, DC, USA, November 9-12, 2015*, pages 1157–1161, 2015.

[31] O. Veksler. Stereo correspondence by dynamic programming on a tree. In *IEEE (CVPR 2005), 20-26 June 2005, San Diego, CA, USA*, pages 384–390, 2005.

[32] S. Xiao, A. M. Aji, and W. Feng. On the robust mapping of dynamic programming onto a graphics processing unit. In *IEEE ICPADS 2009, Shenzhen, China, December 8-11, 2009*, pages 26–33, 2009.

[33] B. Yu, A. Cuzzocrea, D. H. Jeong, and S. Maydebura. On managing very large sensor-network data using bigtable. In *IEEE/ACM CCGrid 2012, Ottawa, Canada, May 13-16, 2012*, pages 918–922, 2012.