Computer Vision and Image Understanding xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Computer Vision and Image Understanding

journal homepage: www.elsevier.com/locate/cviu

Leveraging cost matrix structure for hardware implementation of stereo disparity computation using dynamic programming

W. James MacLean [a,*], Siraj Sabihuddin [a], Jamin Islam [b]

[a] University of Toronto, Department of Electrical & Computer Engineering, Toronto, Canada M5S 3G4
[b] Ryerson University, Department of Electrical & Computer Engineering, Toronto, Canada M5B 2K3

A R T I C L E I N F O

Article history:
Received 19 December 2008
Accepted 17 March 2010
Available online xxxx

Keywords:
Stereo disparity
Dynamic programming
Field programmable gate arrays
Pipeline processing
Real-time systems
Hardware

A B S T R A C T

Dynamic programming is a powerful method for solving energy minimisation problems in computer vision, for example stereo disparity computations. While it may be desirable to implement this algorithm in hardware to achieve frame-rate processing, a naïve implementation may fail to meet timing requirements. In this paper, the structure of the cost matrix is examined to provide improved methods of hardware implementation. It is noted that by computing cost matrix entries along anti-diagonals instead of rows, the cost matrix entries can be computed in a pipelined architecture. Further, if only a subset of the cost matrix needs to be considered, for example by placing limits on the disparity range (include neglecting negative disparities by assuming rectified images), the resources required to compute the cost matrix in parallel can be reduced. Boundary conditions required to allow computing a subset of the cost matrix are detailed. Finally, a hardware solution of Cox's maximum-likelihood, dynamic programming stereo disparity algorithm is implemented to demonstrate the performance achieved. The design provides high frame rate (>123 fps) estimates for a large disparity range (*e.g.* 128 pixels), for image sizes of 640 × 480 pixels, and can be simply extended to work well over 200 fps.

1. Introduction

Dynamic programming is an optimisation algorithm that exploits *optimal substructure* to greatly reduce the time required to compute an optimal solution [1]. It has numerous applications, such as biosequence matching (including DNA sequencing), the Viterbi algorithm for estimation of hidden Markov models (HMMs), and stereo disparity estimation in computer vision. Many problems that can be solved with dynamic programming involve matching data from two streams, such as the left- and right-image pixels from scanlines in stereo images, or problems involving matching string subsequences from two target strings. It should be noted that other problems share this structure, notable examples are the matrix-chain multiplication problem, the Needleman–Wunsch algorithm for biosequence matching, and the Viterbi algorithm for estimating hidden Markov models. This paper explores a dynamic programming approach applied to the stereo matching problem, and gives a sample implementation using FPGAs.

Fast, efficient implementations for dynamic programming problems are of interest, especially in cases where the data rate is high, as it is in image processing. A number of researchers have made attempts to accelerate dynamic programming algorithms using hardware and/or special processors [2–6]. Much of this work has involved protein and DNA sequence matching [2–5] using an "edit-distance" cost function for string comparison [7] which shares the same anti-diagonal parallelism that we exploit in our approach. In [2] the anti-diagonal structure is used to pipeline comparison of multiple target strings against a reference string, but each cost matrix is computed in a row-wise manner. Martins et al. [4] point out the challenges of adapting processing to the changing sizes of the anti-diagonals, and propose a block-wise anti-diagonal approach on a parallel processor architecture to ensure good processor utilisation. Anvik et al. [5] identify the anti-diagonal cost matrix structure as a *design pattern* and introduce a framework generator that minimises the development effort required to adapt this design pattern to specific applications (they demonstrate three such applications) running on a small array of four processors. They use a block-wise approach similar to that of [4]. Two of these approaches involve implementation on FPGAs [3,6], with both being tested in simulation only. The approach of [3] uses multiple FPGAs,[1] but does not re-map the cost matrix as we do, resulting in half of the processing elements being idle on any given cycle. Hoang and Ayala-Rincón et al. [3,6] both use systolic arrays in their processing architecture, although it is

* Corresponding author.
E-mail address: james.maclean@utoronto.ca (W.J. MacLean).
URL: http://www.wjamesmaclean.net/ (W.J. MacLean).

[1] At the time their paper was written, FPGA devices had much smaller capacity

1077-3142/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.cviu.2010.03.011

pointed out that adapting such arrays for computing only a diagonal-band of the cost matrix (as we do by limiting the disparity range D) is challenging.

Stereo matching has been extensively studied by the computer vision community. Detailed reviews of existing algorithms have, thus, also been published by a number of researchers. Gong et al. [8] provide an overview of general approaches that utilise sparse, dense, volumetric or level-set algorithms. Furthermore, Seitz et al. [9] provide a review of multi-view matching methods. Some image registration techniques also utilise sparse, feature based matching and are discussed by Zitova and Flusser [10].

The dynamic programming solution presented in this paper is the ‘‘Dynamic Programming Maximum Likelihood'' (DPML) stereo disparity algorithm by Cox et al. [11]. It is a global and dense disparity estimation algorithm—Scharstein and Szeliski [12] and Brown et al. [13] provide extensive discussions and comparisons of similar dynamic programming approaches. These comparisons demonstrate that dynamic programming provides accurate estimates that compete well even with the best of existing matching methods. More accurate approaches do exist, however, they tend to operate at significantly lower speeds (see [14]). Closely related to the studies by Scharstein and Brown, Lu et al. [15] provide a further survey on cost aggregation for matching problems.

In current literature, the most common approaches towards FPGA hardware stereo matching are based on correlation or area-based methods. Of these, the most common methods make use of SAD aggregated cost functions applied to pixel intensities. As shown by Scharstein and Szeliski in [12] SAD correlation approaches do not provide very accurate disparity estimates. Typically these implementations make use of line buffering to align pixels in a stereo pair for parallel windowing computations. Works by Miyajima and Maruyama [16], Hariyama et al. [17], Perri et al. [18], Mitéran et al. [19] [7] and Han et al. [20] all utilise such buffering. Both Perri et al. [18] and Mitéran et al. [19] observe that neighbouring windows of SAD computations use many of the same values. These values are stored and re-used as required. Hariyama et al. [17] define two levels of parallelism to perform coarse to fine refinement of disparities within localised regions, and thus improve the accuracy. Both Hariyama et al. [17] and Simhadri et al. [21] utilise adder trees to perform cost and comparison computations. It is worth noting that adder trees are primarily useful in situations such as windowing—they do not typically apply to dynamic programming type solutions. Lee et al. [22] present an FPGA-based SAD stereo algorithm running at 31 fps for 640×480 images with a disparity of 64 pixels, although no quantitative accuracy results are given. Most of these implementations produce results at speeds of approximately 30 fps, although some go as high as 150 fps on smaller images. Typically, papers that present these solutions do not present an analysis of accuracy of their algorithms.

Additional hardware methods use phase based correlation. Díaz et al. [23], Darabiha et al. [24] and Masrani and MacLean [25] provide examples of these approaches implemented in hardware. Phase based methods have the advantage of producing significantly better results than SAD algorithms, obtaining results that compete well with dynamic programming solutions. The problem with these methods lies in the square root computations required—these computations are difficult to implement in hardware and come with an associated high resource cost, especially when implemented in parallel. Both Darabiha et al. [24] and Masrani et al. [25] provide an implementation that uses multi-scale Locally Weighted Phase Correlation (LWPC). Like SAD based implementations these phase based approaches achieve approximate 30 fps performance. Díaz et al. [23] demonstrate an algorithm that achieves over 200 fps performance but does so by reducing the search range to only four neighbouring pixels.

A final approach that produces very high frame rates (over 200 fps), on par with the approach presented in this paper, makes use of the census algorithm (see Woodfill et al. [26]). The census algorithm computes costs within a pixel neighbourhood using the census transform—the transform essentially utilises hamming distance based comparisons. The high frame rates and very simple computations associated with this approach come with poor accuracy. Murphy et al. [27] present a ‘‘low-cost'' implementation of the census algorithm on a Xilinx Spartan-3 FPGA, with performance of 40 fps and a disparity range of 20 pixels, with no accuracy results presented.

To date, no hardware implementations of dynamic programming algorithms appear to have been explored by the vision community. Furthermore most existing solutions produce relatively low frame rates of approximately 30 fps at resolutions that are typically lower than 640×480 pixels. They achieve higher frame rates by sacrificing on accuracy, resolution or disparity search range. This paper explores a hardware based dynamic programming solution that achieves high frame rate performance with no compromise on accuracy and limited compromise on disparity search range or resolution. The solution provides good scaling characteristics relative to SAD and phase based correlation approaches.

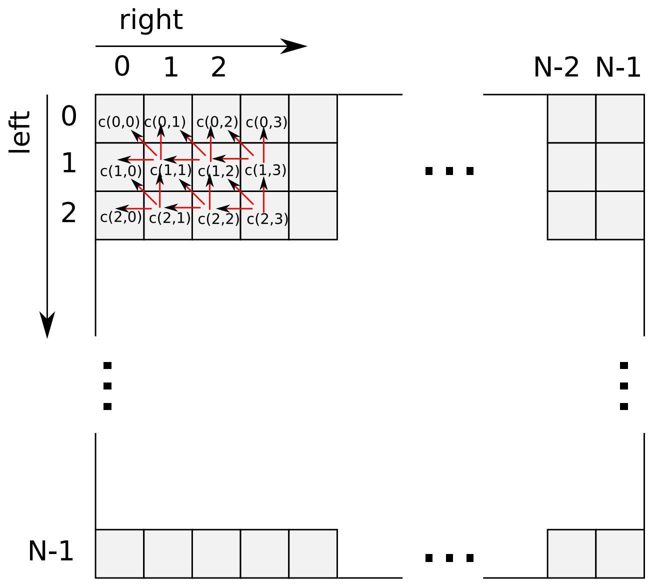
The rest of this paper is structured as follows. In Section 2 we describe the structure inherent in a dynamic-programming cost matrix and how this structure can be exploited to design efficient hardware. In Section 3 we describe a hardware implementation of a dynamic programming algorithm for stereo disparity estimation, showing how the cost matrix structure leads to a fast yet efficient hardware design. Finally, in Section 4, we give performance results for the stereo hardware implementation. Further technical details of the implementation can be found in [28–30].

2. Cost matrix structure

When solving a dynamic programming problem using a bottom-up approach, a typical approach is to build a cost matrix, denoted by C. In this matrix element C_{ij} represents the optimal cost of associating two data elements $I_L(i)$ and $I_R(j)$. This paper will concentrate on computing stereo disparity using these data elements. Note that $I_L(i)$ represents the ith pixel from the left scanline and $I_R(j)$ the jth pixel from the right scanline. Both $I_L(i)$ and $I_R(j)$ lie within the same scanline (vertical position) in the left and right images.

In computing the cost matrix, an element C_{ij} depends on those neighbouring elements with lower indices. In computing an optimal cost for element C_{ij}, one assumes that optimal costs have already been computed for the elements $C_{i-1,j}$, $C_{i,j-1}$ and $C_{i-1,j-1}$, and computes the optimal cost for the current element in terms of these. These cost dependencies are shown in Fig. 1. In a traditional software implementation it is typical to do this row-wise with a nested loop structure, which computes elements left-to-right along each row, assuming the results from the previous row are complete.

This approach assumes that all the data required to compute an entire row of C is already present. In the case of stereo disparity computations, this means that the cost value for a particular combination of right/left pixel positions (i and j) in a scanline cannot be computed unless all previous dependant input pixels have been received and their respective costs computed. Even if, for a particular problem, all the data are available at the outset, the computations must be done sequentially as each element on the row depends on the ones before it. While this is not a problem for software designed to run on a serial processor, it hampers efforts to parallelise cost computations in hardware. It does allow for partial storage of C, as once a row of C has been computed, only the current row is



right
0 1 2
N-2 N-1
left
0
1
2
N-1
c(0,0) c(0,1) c(0,2) c(0,3)
c(1,0) c(1,1) c(1,2) c(1,3)
c(2,0) c(2,1) c(2,2) c(2,3)

Fig. 1. Computing a cost matrix element assumes that the elements immediately left, above, and above-left have already been computed.

required for computing the next row. Since the optimal-cost path can be summarised in a match matrix using fewer bits per element, the cost matrix C is responsible for most of the memory requirements of the algorithm. Therefore, the ability to store only the current row of the cost matrix can result in significant resource savings in a hardware implementation.

The row-wise approach is not the only choice, however. It is also possible to perform cost computations along anti-diagonals, that is along elements C_{ij} such that $i + j = k$ where k is a positive constant, as shown in Fig. 2.

This approach has several advantages. First, computing the elements of a given anti-diagonal only requires storage of a fixed number of preceding anti-diagonals—two anti-diagonals in the case of stereo and longest subsequence matching, although the matrix-chain multiplication problem requires more than just the two preceding anti-diagonals. While this can be up to twice as much storage as just storing the preceding row, it is still far cheaper than storing the entire cost matrix. If only a limited range of elements are computed, as in the case of limiting the disparity range in stereo, storage requirements are further reduced. Second, and most important, the computations of the cost elements along a given anti-diagonal can be computed independently of each other, and thus they can be computed in parallel (see Fig. 2).

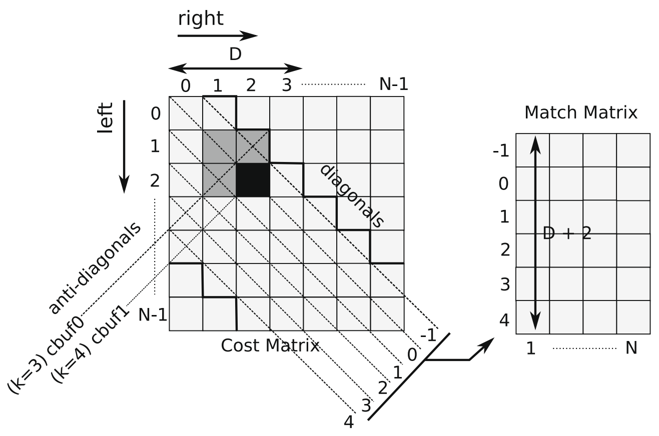


Fig. 2. Cost and match matrix structure for stereo matching using dynamic programming, for images of width N pixels, and a disparity range of $0 \ldots D-1$ pixels ($D = 4$ in this diagram).

This approach does not require all the input pixel data to be available prior to computing a given anti-diagonal; rather it just requires pixel data $\{I_L(i), I_R(i)\}_{i=0}^{k}$ to compute up to the kth anti-diagonal. In what follows, the notation AD0 refers to the anti-diagonal with $k = 0$, AD1 the anti-diagonal with $k = 1$, *etc.* For example, AD0 can be computed knowing only $\{I_L(0), I_R(0)\}$, AD1 can subsequently be computed once $\{I_L(1), I_R(1)\}$ arrive, AD2 can then be computed once $\{I_L(2), I_R(2)\}$ have arrived, and so on.[2] This allows a parallel approach, where the intermediate results (the cost elements of anti-diagonals) leave the pipeline at the same rate that the data flow in. This is significant as it means that the entire cost matrix can, potentially, be computed in the time it takes for all the data to arrive.

A few facts are worth noting. If the cost matrix is $N \times N$ (as in the case of stereo for images of width N pixels), then there are $2N - 1$ anti-diagonals. Also, for a given anti-diagonal, the difference in the indices $i - j$ for all elements C_{ij} along the anti-diagonal will either all be even, or they will all be odd. In the case of stereo, this suggests that a given anti-diagonal represents even-valued disparities, or odd-valued disparities, but not both. This means that two adjacent anti-diagonals encode a full set of disparities (both odd and even) in a given range. Further, if we are only interested in elements for which $i \geqslant j$ (as in the case of rectified stereo images), then only cost matrix elements on or below the main diagonal (see Fig. 2) need be considered. Finally, if we assume that $i - j < D$, as in the case of setting an upper limit D on the range of allowed disparities, then only a band of sub-diagonals in C need be computed. These two limitations, namely $i - j \geqslant 0$ and $i - j < D$, allow hardware resources to be conserved, always an important consideration in FPGA implementations. The elements C_{ii} along the main diagonal represent the optimal costs for a disparity of zero at the ith pixel.

3. Example hardware implementation

In this section a sample FPGA implementation is presented for the DPML stereo algorithm of [11] (shown in Listings 1 and 2 for reference).

The comparisons produce cost estimates for each pixel location based on the cost functions given by Eqs. (1) and (2). $I_L(x)$ and $I_R(x)$ are pixel values at position x in a scanline, while d is a disparity within range: $0 \ldots D-1$. It is worth noting that the occlusion cost in Eq. (2) algorithm is analogous to the insertion/deletion costs in the edit–distance algorithm, mentioned in Section 1.

$$NOC(I_L(x), I_R(x+d), \sigma) = \frac{(I_L(x) - I_R(x+d))^2 \sigma^2}{4} \tag{1}$$

$$OC(P_d, \sigma^2, \phi) = \log \frac{P_d \phi}{(1 - P_d)} \sqrt{\frac{2\pi}{\sigma^2}} \tag{2}$$

In Eqs. (1) and (2), σ represents a noise variance associated with the image sensor, P_d is the probability of a given pixel being visible in both views (*i.e.* not being occluded in the other view) and ϕ is the field of view of the camera.

After cost values for all pixel locations are stored in an $N \times N$ cost matrix, C, the backward pass initiates (see Listing 2). Note that N is the number of pixels in an image scanline. An optimal path is traced out through an $N \times N$ match matrix (M) to compute the stereo disparity estimates. Please refer to [11] for further details.

[2] In many stereo applications the left and right cameras are synchronised, so the pixel data arrive in pairs.

```
% Initialise match and cost matrices
for c=1:N
    M(c,1:c)=1; M(c,(c+1):N)=2;
    C(c:N,c)=(((c-1):(N-1))*OC);
    C(c,c:N)=(((c-1):(N-1))*OC);


% For each pixel in a row compute NOC, OC
for cl=2:N
    for cr = cl:(cl-D)
        min1 = C(cr-1, cl-1) + NOC;
        min2 = C(cr-1, cl) + OC;
        min3 = C(cr, cl-1) + OC;
        C(cr,cl) = min(min1, min2, min3) = cmin;

        if (min1==cmin)     M(cr,cl)=0; % No occlusion
        elseif (min2==cmin) M(cr,cl)=1; % Right occlusion
        elseif (min3==cmin) M(cr,cl)=2; % Left occlusion
```

Listing 1. The Cox et al. DPML algorithm [11], forward-pass, in Matlab-like notation. Cost and match computations for each pixel location in a scanline, over a disparity range, *D*.

```
p = N; q = N;
while (p≠1 && q≠1)
    if M(p,q)==0
        DISP(q)=abs(p-q);
        OCC(q)=0;
        p--; q--;
    elseif M(p,q)==1
        OCC(q)=1; p--;
    elseif M(p,q)==2
        OCC(q)=1; q--;
```

Listing 2. The Cox DPML algorithm, backward-pass. Computation of optimal disparity values for pixels, in a scanline, relative to the left image of the stereo pair. Disparity/occlusion computations makes use of the cost/match computations in 1.

The FPGA implementation described in this paper is based on an earlier system [31], but leverages the cost matrix structure to dramatically improve performance. FPGA implementations of computer vision algorithms have been increasing in recent years, and in general [32] FPGAs are being utilised increasingly in high-performance computing applications. In image processing, considerable speedup is possible due to parallelisation of operations that would normally occur sequentially in software. The implementation is pipelined in the sense that it completes the cost-matrix computation in the time it takes for the pixels from both left- and right-scanlines to be received. The computed disparity is limited to a range of $0 \ldots D-1$, although it will be seen that this is merely a consideration for resource usage on the FPGA, and D can be set to the image width if desired, assuming the FPGA resources are sufficient. The choice of D only minimally affects the total processing rate. This minimal effect is a result of the initial delay required to align left and right-image pixels for the first anti-diagonal computation. Barring this small delay, an increase in D simply adds more hardware in parallel. The implementation assumes the images are pre-rectified, thus allowing negative disparities to be discarded. In order to confine the final, optimal pairing of pixels (path) to lie within this range, suitable boundary conditions are required for the diagonal above the main diagonal and below the $D/2$-th sub-diagonal. These conditions are described below in Section 3.3. Costs along a given anti-diagonal are computed in parallel, so D sets an upper limit of $N_{AD} = D/2 + 1$ on the number of parallel computation units required. D is assumed to be even.

The overall architecture of the final implementation (referred to as DPMLHW(I) in Section 4) is shown in Fig. 3, although in Section 4 results are shown for several intermediate design stages as well. In the subsequent sub-sections, important aspects of the design are described in detail.

3.1. Buffering incoming pixels

One important aspect in computing the cost matrix entries involves aligning pixels as they arrive from the left- and right images in a manner appropriate for computing costs along the current anti-diagonal. Two buffers, LBUF and RBUF as shown in Fig. 4, are used for this. For each left–right pair of pixels, two anti-diagonals (even and odd) are processed. Each time an even-numbered anti-diagonal is processed, LBUF is shifted left by one, and a new left-image pixel is inserted in the left-most available slot. Each time an odd-numbered anti-diagonal is processed, a new right-image pixel is right-shifted into RBUF. In both cases, the difference in pixel intensities of corresponding valid LBUF and RBUF elements are passed to the parallel cost computation units described in the next section.

During start up, pixels are also added at the left-most empty slot of LBUF on odd anti-diagonals until LBUF is full. On alternating even and odd anti-diagonals, the last and first corresponding elements of the buffers are marked as invalid to indicate that boundary conditions should be applied instead. In addition, empty buffer elements in both LBUF and RBUF are marked invalid at the beginning and end of the scanline. An example of the contents of both LBUF and RBUF during processing of the first four scanlines is shown in Fig. 5.

3.2. Computing cost matrix elements

Cost matrix values are stored in three arrays of length N_{AD}, where the first two arrays (CBUF0 and CBUF1) contain the preceding two anti-diagonal's cost values, and the third array, PMIN/CMUX, is working memory for the current anti-diagonal, as shown in Figs. 3 and 6. Fig. 6 shows the parallel-load operation of the cost array that allows discarding anti-diagonals that are no longer needed. CBUF0 holds the cost elements for the anti-diagonal two-before the one currently being computed, and CBUF1 for the anti-diagonal one-before.

Each cost matrix element is computed based on the non-occlusion cost (NOC) function of the pixels $I_L(i)$ and $I_R(j)$ (see Eq. (1) and block PNOC in Figs. 3 and 6). While it might seem obvious to put this functionality in a lookup table to save resources, having N_{AD} parallel accesses to such a table is not feasible for realistic values of D (for example, $D = 128$), so separate computations are carried out for each C_{ij}. It is straightforward to re-scale the cost values to reduce the computation of NOC to one subtraction and one multiplication. For each element, the occlusion costs based on $C_{i-1,j}$ and $C_{i,j-1}$ are compared to the result of adding the NOC to $C_{i-1,j-1}$, and the minimum value is assigned to C_{ij} (see block PMIN/CMUX in Fig. 3). If the NOC is chosen, 0 is written to the corresponding element of the match matrix (see Section 3.4), otherwise 1 or 2 is stored to indicate a left- or right-occlusion, respectively.

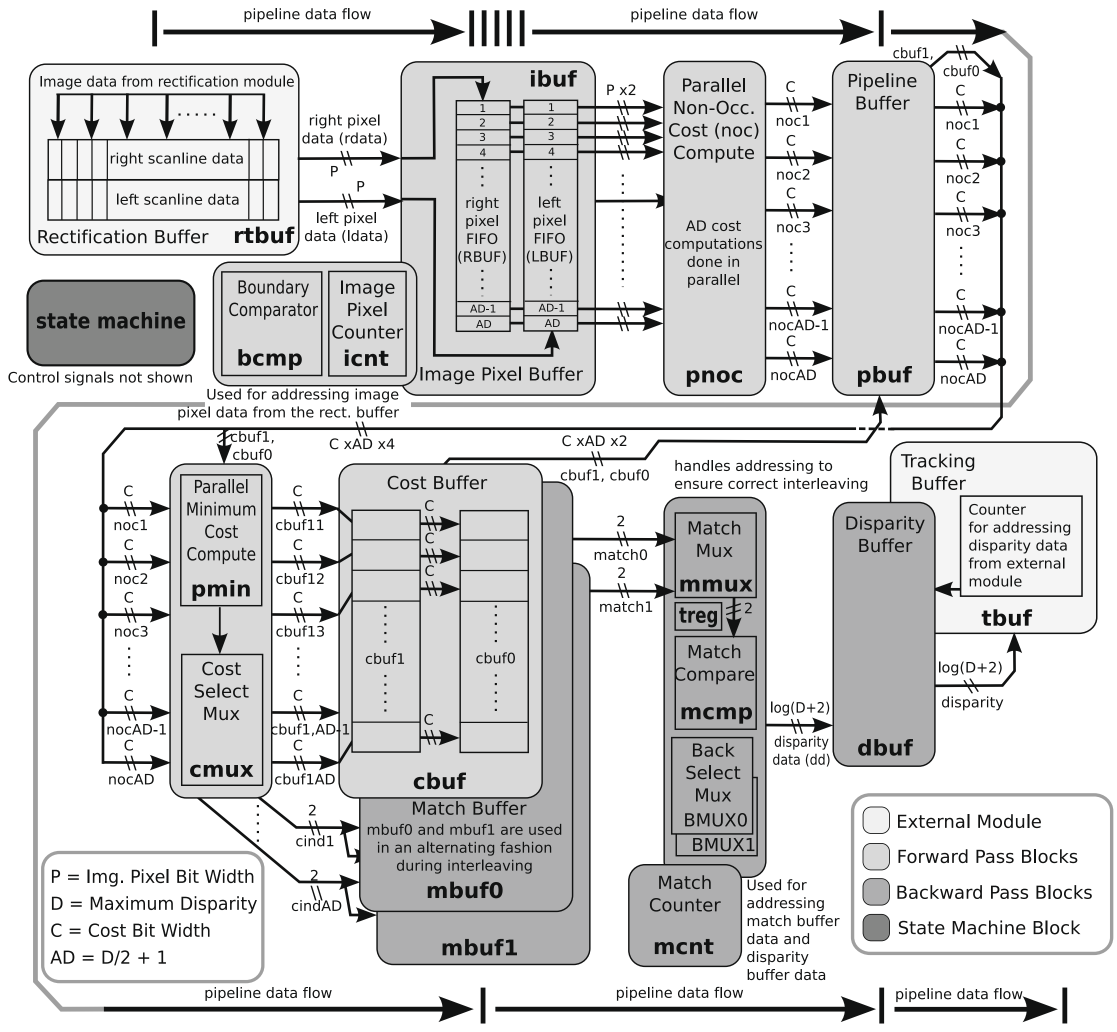


Fig. 3. Schematic diagram of the final hardware implementation of the DPML algorithm, for disparity range $0 \ldots D-1$.

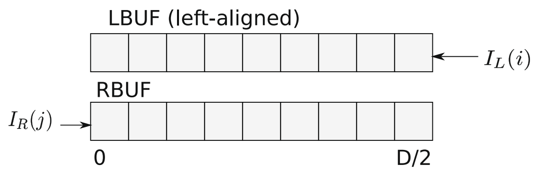


Fig. 4. Buffers LBUF and RBUF for storing incoming pixels, shown for the case $D = 16$. LBUF is a left-aligned shift register that accepts pixels on every anti-diagonal, but only shifts out before accepting input on even scanlines. RBUF is also a shift register, but it only accepts input on odd anti-diagonals.

Once costs have been computed for the current anti-diagonal, and corresponding match matrix entries made, the first row of the cost array (CBUF0) is discarded and the second and third rows shifted up to make way for cost values in the next anti-diagonal.

3.3. Boundary conditions

Regardless of the value of D, boundary conditions are required along the first row and first column of the cost matrix, as described in Cox's algorithm. If $D < N$ then boundary conditions are also required at either end of the array PMIN/CMUX. For odd anti-diagonals, the first element in the cost array represents the super-diagonal (the first diagonal above the main diagonal, indicating a disparity of -1): this element is assigned the cost of the preceding super-diagonal element plus a fixed occlusion cost. Similarly, on even anti-diagonals, the last element in the cost array is assigned the cost of the last element of the previous even anti-diagonal, plus the fixed occlusion cost. For the initial $D/2$ anti-diagonals, where there are less than N_{AD} valid entries in the anti-diagonal, the cost of the last valid entry of even anti-diagonals is just $i \times \text{OC}$, where OC is the fixed occlusion cost. In all cases, bounding cost matrix elements will have a cost assigned to them as specified in the initialisation phase of Listing 1.

The boundary conditions are important for propagating cost values that lie within the disparity range being estimated. It should be noted that the optimal path will never wander outside this range as the corresponding match matrix entries are automatically marked as occluded, meaning the backtracking algorithm will (at worst) follow along these boundaries, but never cross them.

3.4. Match matrix and backtracking

As each cost matrix element is computed, corresponding values are written into a match matrix, M, which is subsequently used to backtrack through the optimal path once all cost matrix values have been computed. The value 0 is stored if the minimum cost was achieved through the NOC function, else 1 or 2 is stored to indicate a left- or right-occlusion, respectively. Each element of M only requires two bits to store, so M uses little in the way of resources. While M is nominally $N \times N$ (see Listings 1 and 2), if the

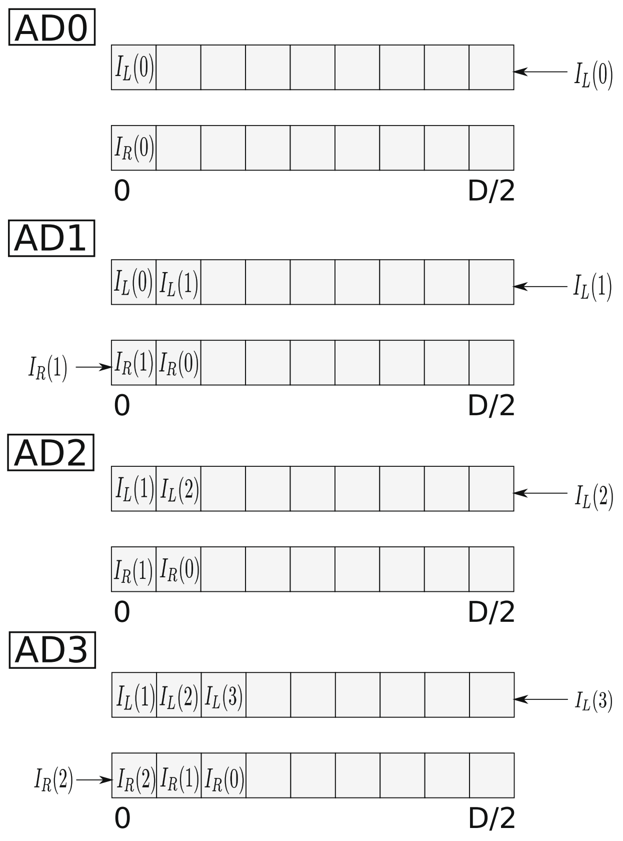


Fig. 5. The contents of LBUF and RBUF are shown for the first four anti-diagonals. Both buffers are cleared at the start of a scanline, and $I_R(0)$, the first pixel from the right scanline, is inserted into RBUF before processing starts on AD0. *AD0*: LBUF is shifted left and $I_L(0)$ is inserted in the left-most available position. Adjacent elements are compared to compute NOC values (see Fig. 6). *AD1*: The value $I_R(1)$ is shifted into RBUF and value $I_L(1)$ is placed in the left-most available position in LBUF, and the NOC values for AD1 are computed. Insertion of pixels into LBUF on odd anti-diagonals is only performed until LBUF fills. *AD2*: LBUF is shifted to the left, and then value $I_L(2)$ is added to the left-most available position, then NOC values for AD2 are computed. *AD3*: Value $I_R(2)$ is inserted into RBUF, shifting existing values to the right. Value $I_L(3)$ is added to the left-most available position in LBUF, and then NOC values for AD3 are computed.

disparity range is limited to $0...D-1$ then only $N \times D + 2$ elements need to be allocated to store *M*s values, re-mapping *M* as shown in Fig. 2. The backtracking algorithm has been modified to generate the appropriate indices for the reduced-size match matrix. In Fig. 3, *M* is represented by the buffers MBUF0 and MBUF1 (see Section 3.7).

3.5. State machine implementation

A state machine, depicted in Fig. 7, is used to control the computation of the cost- and match-matrix elements, and to perform the backtracking that produces the final disparity map. The state `reset` initialises the cost array to store zeros, initialises the appropriate boundary conditions in the match matrix and waits for the first left and right pixels to arrive. Control then passes to the `skip` state, which waits for enough pixels to enter LBUF and RBUF so that computations can begin on anti-diagonal #2. At this point control alternates between states `ad_even` and `ad_odd`, which apply appropriate boundary conditions, compute costs for the current anti-diagonal, and update the appropriate match matrix elements. Since $2N + 1$ is always odd, `ad_odd` is always the current state when the cost computations are completed. Control passes to the `backtrack` state, which works backwards through *M* generating the optimal disparity and occlusion maps. Once backtracking is complete, control returns to `reset` to await arrival of the next scanline. A more detailed description of this process, with reference to Fig. 3, follows.

3.5.1. Forward pass

During the forward-pass, left and right image data for a particular scanline is obtained from an external memory location (RTBUF). This memory location is used to store rectified image data. A counter, ICNT, is utilised to address this memory. A boundary comparator, BCMP, ensures that the counter is appropriately incremented, decremented or reset so as to remain within the bounds of the available image dimensions. During an initialisation phase, image pixels are retrieved from RTBUF and pushed in from opposing directions into LBUF and RBUF, as described in Sections 3.1 and 3.2. NOC computations take place in parallel in the asynchronous PNOC block and the results of these parallel computations are stored in a pipeline buffer (PBUF). The minimum of the three costs is computed by PMIN, and an associated index, representing the minimum cost, is generated by the cost multiplexer

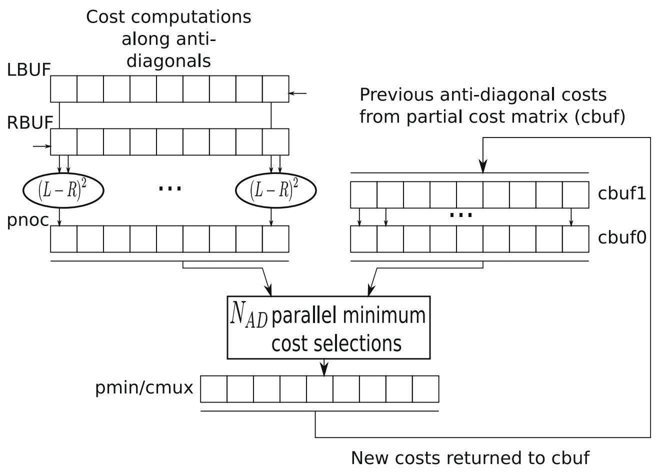

Cost computations along anti-diagonals
LBUF
RBUF
$(L-R)^2$
...
$(L-R)^2$
pnoc
Previous anti-diagonal costs from partial cost matrix (cbuf)
cbuf1
...
cbuf0
N_{AD} parallel minimum cost selections
pmin/cmux
New costs returned to cbuf

Fig. 6. Cost computation dataflow: non-occlusion costs (NOC) are computed for adjacent elements in LBUF and RBUF and placed in PNOC. These values are combined with costs from two previous anti-diagonals stored in CBUF, and the result placed in PMIN/CMUX. Finally, values are transfered from CBUF1 to CBUF0 and the new values from CMUX are transfered to CBUF1.

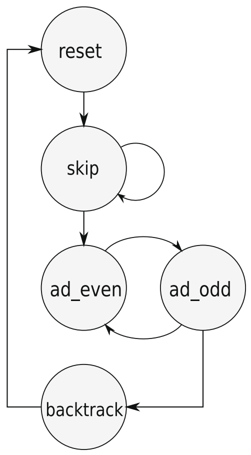


Fig. 7. Initial concept for state machine used to control cost and match matrix computation. This state machine is refined in Figs. 8 and 10, the latter representing the final state machine design.

(CMUX). Both PMIN and CMUX operate asynchronously. At the following clock cycle these final computed anti-diagonal costs (computed in parallel for each position in the anti-diagonal) are stored in CBUF. On the incoming clock cycle, the final costs are also stored into the match buffer (MBUF0 or MBUF1, see Section 3.7) for backward-pass computations. The match buffer is indexed by addresses generated by the MCNT counter.

Black horizontal arrows and vertical bars at the top and bottom of the block diagram (Fig. 3) indicate the positions of pipeline registers. These registers allow cost computations to occur at the same rate as the rectified image pixels are generated. It should be noted, though, that a small initial lag exists. This lag is required to fill the pipeline for the first anti-diagonal computations.

3.5.2. Backward pass

Following the forward-pass, a backward-pass is used to read data from MBUF0 or MBUF1 (see Section 3.7). The hardware backtracks through this buffer for the generation of a final set of disparity values at each location in a given scanline. The match comparator, MCMP, is used to compare addresses generated by the match counter. Address comparisons are utilised to determine the next most appropriate location for the optimal path in the match buffer.

Since the match buffer is addressed not only by a pixel location in the scanline, but also a disparity value, these comparisons directly affect the final disparity results. The disparity is given by MCNT and stored in the disparity buffer, DBUF, when an appropriate comparison is made by MCMP. At the end of the backward-pass, this disparity value can be retrieved by an external module, for tracking tasks (TBUF) or direct display. The data can of course also be retrieved by directly accessing it as it is generated by the backward pass.

Since the backward- and forward-pass go through two distinct pipelines, a back select multiplexer (BMUX0 and BMUX1) are used to re-route control signals to ensure appropriate synchronisation between the backward phase's address, data and control logic.

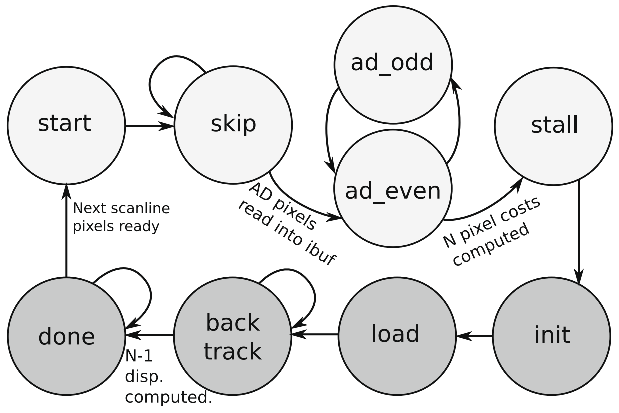


Fig. 8. State machine used to control cost and match matrix computation in the DPMLHW(P) hardware design. Light-coloured states represent the forward-pass, in which the match matrix elements are computed, and darker-coloured states represent the backward pass that computes the optimal path through the match matrix.

3.6. Pipelined cost computation

Without PBUF or the interleaving described in Section 3.7, the design performs at about 53 fps for 640×480 images with $D = 128$. While this is very respectable performance, analysis shows that the computation of the NOC (PNOC), together with the minimum cost computation (PMIN), has a larger combinational delay than any of the other elements of the design. This suggests that use of PBUF will break up this delay, improving the performance even further to 63 fps. For the cost of a small initial delay (one clock cycle) to fill the pipeline register, throughput is substantially increased. The addition of PBUF between PNOC and PMIN (Fig. 3) allows these computations to proceed in parallel, thus reducing the net combinational delay. As a result, additional registers are also required in CBUF. Cost computations from the previous clock cycle can be re-used by feeding them back into PBUF. During the backward pass, to prevent read delays at the RAM (MBUF), data must be pre-fetched from several different addresses simultaneously.

The use of pipelining requires modification to the controlling state machine, shown in Fig. 8. The addition of the `stall` state at the end of the forward-pass ensures that the last pixel computation has time to reach the end of the forward pipeline. This is

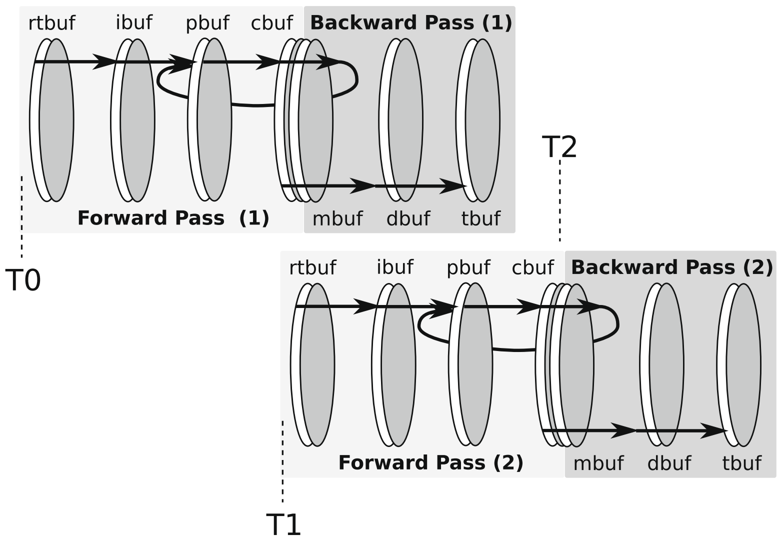


Fig. 9. Throughput can be increased by performing the backward pass for a given scanline at the same time as performing the forward-pass for the following scanline. This figure illustrates the backward pass for scanline 1 occurring in the same time period (T1–T2) as the forward pass for scanline 2. Two separate match-matrix buffers are required to be able to do this.

necessary due to the one clock cycle delay incurred when initially transferring data into the pipeline registers in PBUF. During the backward pass, MBUF's synchronous read operations have a one clock cycle latency from the time that an index address is placed on the address lines. The `init` state ensures that this latency is taken into account for the first read operation. Since the next read address for MBUF is determined by the currently read data, it is necessary to pre-fetch the data for all potential next address candidates. Without this pre-fetch, a one clock cycle penalty would be incurred for every read operation executed by MBUF. The `load` and `backtrack` states implement this pre-fetch and compute disparity values for all pixel locations in the current scanline.

3.7. Forward-/backward-pass interleaving

A final improvement can be made to further improve performance. The design, as described thus far, proceeds to execute the backward pass before beginning a new forward-pass. This is limiting, and is not necessary. Scanline throughput can be increased by interleaving forward and backward phases of the DPML algorithm execution as conceptually shown in Fig. 9 and more formally in Fig. 10.

After the execution of the first forward-pass, the next scanline's forward-pass can proceed concurrent to the first backward-pass. In order to facilitate this, the match buffer is duplicated into two identical units MBUF0 and MBUF1 (Fig. 3). During a particular for-

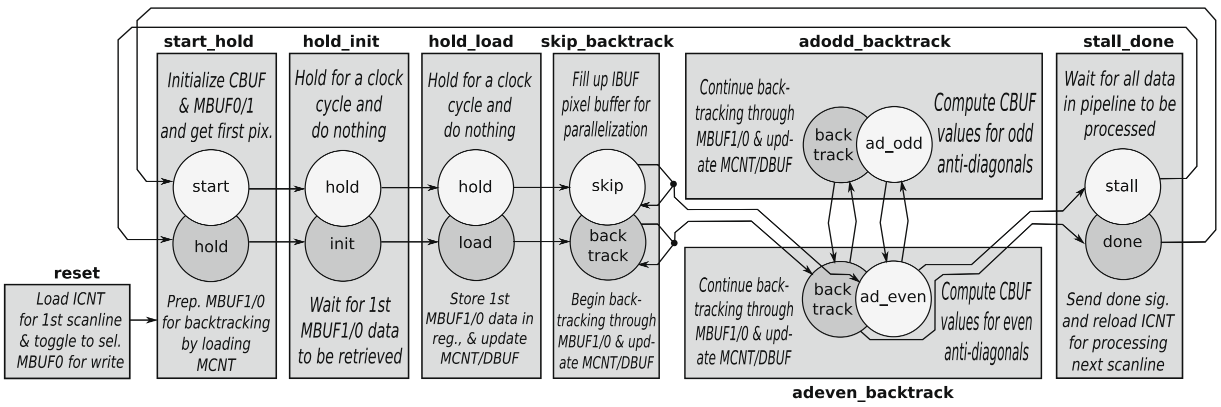

start_hold
Initialize CBUF & MBUF0/1 and get first pix.
start
hold
Prep. MBUF1/0 for backtracking by loading MCNT
hold_init
Hold for a clock cycle and do nothing
hold
init
Wait for 1st MBUF1/0 data to be retrieved
hold_load
Hold for a clock cycle and do nothing
hold
load
Store 1st MBUF1/0 data in reg., & update MCNT/DBUF
skip_backtrack
Fill up IBUF pixel buffer for parallelization
skip
back track
Begin back-tracking through MBUF1/0 & update MCNT/DBUF
adodd_backtrack
Continue back-tracking through MBUF1/0 & update MCNT/DBUF
back track
ad_odd
Compute CBUF values for odd anti-diagonals
Continue back-tracking through MBUF1/0 & update MCNT/DBUF
back track
ad_even
Compute CBUF values for even anti-diagonals
adeven_backtrack
stall_done
Wait for all data in pipeline to be processed
stall
done
Send done sig. and reload ICNT for processing next scanline
reset
Load ICNT for 1st scanline & toggle to sel. MBUF0 for write

Fig. 10. Revision of state machine in Fig. 8 that interleaves the forward-/backward-passes. Notice the addition of controls to select which match-matrix buffer is used by each pass, as well as the synchronisation between the states of the two passes. The labels MBUF1/0 and MBUF0/1 indicate mutually exclusive memory access between corresponding sub-states.

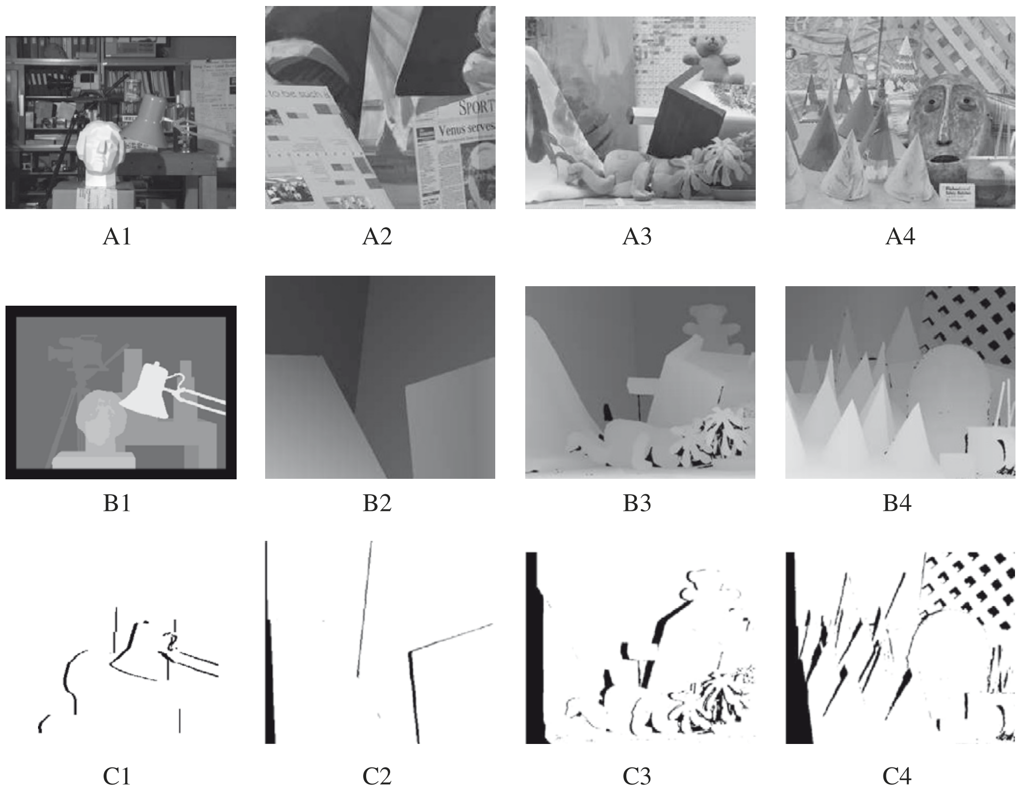


Fig. 11. Tsukuba, Venus, Teddy and Cones data sets and their ground truths. Sub-images A1–A4 show the left image acquired by the stereo camera system. Sub-images B1–B4 and C1–C4 show ground truth disparity and occlusion maps respectively. These standard data sets are used for evaluating accuracy of stereo correspondence results [33–35].

ward-pass, one of these match buffers (MBUF) is utilised for storage of cost indices while the other is utilised for the backward-pass disparity computations of the preceding forward-pass. The match multiplexer, MMUX, is responsible for toggling the MBUFs for these operations.

To prevent write conflicts and ensure that all data is correctly synchronised, a set of two back select multiplexers (BMUX0 and BMUX1, Fig. 3) are utilised. Each of these multiplexers alone is responsible for synchronising control signals between the two distinct pipelines of the forward and backward pass in one particular scanline computation. However, together these multiplexers flip flop two sets of control signals between the two match buffers such that they operate in a mutually exclusive (and interleaved) manner. A register, TREG, provides a stable and synchronised toggle (address) signal for MMUX so that the results from MBUF can be read and routed correctly to the match comparator, MCMP. The current match buffer, selected by MMUX, is associated with the backward phase of execution. It does so with a one clock delay (via TREG) relative to the BMUX0 and BMUX1 addressing signal—this allows retrieval of data from the synchronous match buffer memory.

Table 1
Accuracy rankings, root mean squared error and percent bad matching pixels for four standard data sets from Scharstein et al. [33,34]: Tsukuba, Venus, Teddy and Cones. Note that accuracy rankings are determined by evaluation tools from [35]. Lower values indicate better performance. Also note that four algorithms are compared: (1) DPMLHW, Dynamic Programming Maximum Likelihood in Hardware. (2) DPML, Dynamic Programming Maximum Likelihood in software. (3) CORR, Correlation with 11×11 window size. (4) SSD, Sum of Squared Difference with 11×11 window. (CORR and SSD implementations from [35].)

Algorithm	Avg. rank	RMS error			
		Tsukuba	Venus	Teddy	Cones
DPMLHW	36.8	0.74	1.13	1.07	1.12
DPML	36.8	0.74	1.13	1.07	1.12
CORR	37.0	1.36	1.07	2.39	2.17
SSD	38.0	2.45	3.77	6.95	5.31
		% Bad Pixel Match			
		Tsukuba	Venus	Teddy	Cones
DPMLHW	36.8	2.81	4.75	3.44	3.84
DPML	36.8	2.81	4.75	3.44	3.84
CORR	37.0	6.09	3.69	8.93	6.52
SSD	38.0	12.37	13.75	27.29	17.24

4. Results and discussion

This section provides a summary of the performance of the example hardware implementation. Performance and accuracy of correspondence results are presented in the context of existing software algorithms, and also compared to other hardware implementations in the literature. Furthermore, the accuracy results are based on standard stereo data sets and quality metrics compiled and used, respectively, by Scharstein et al. [33–35]. All hardware results are reported for a Xilinx XC2VP100 device—this is a moderately large FPGA in the Virtex 2 family, but is by no means the largest or fastest FPGA device available (currently Xilinx is producing Virtex 6 devices with much greater capacity and speed). FPGA development and testing was done using Xilinx's ISE (versions 7.1, 8.1 and 9.1), ChipScope Pro (versions 8.2i and 9.1i), and ModelSim SE 6.2f.

The "ground-truth" datasets used are shown in Fig. 11. For each image set, both left and right images from a stereo pair are given, along with a manually determined ground-truth disparity map. Further, left- and right-occlusion maps are given so that algorithm accuracy can be compared separately in occluded and non-occluded image regions. The images are chosen to be challenging in a variety of ways, including regions of low texture and changes in imaged object sizes between images. These datasets are part of an effort to provide standard datasets for evaluating stereo disparity algorithms, and are widely used by the computer vision community.

4.1. Accuracy

Disparity results generated in the hardware implementation are identical to the equivalent software implementation. Accuracy of these results have been evaluated by direct comparison of the estimated disparities from the hardware and software implementations, so it is no surprise that the root mean squared and bad matching pixel metrics give identical results. It is noted that results are superior to the Sum of Squared Difference (SSD) and Correlation algorithms (based on implementations from [35]) which are

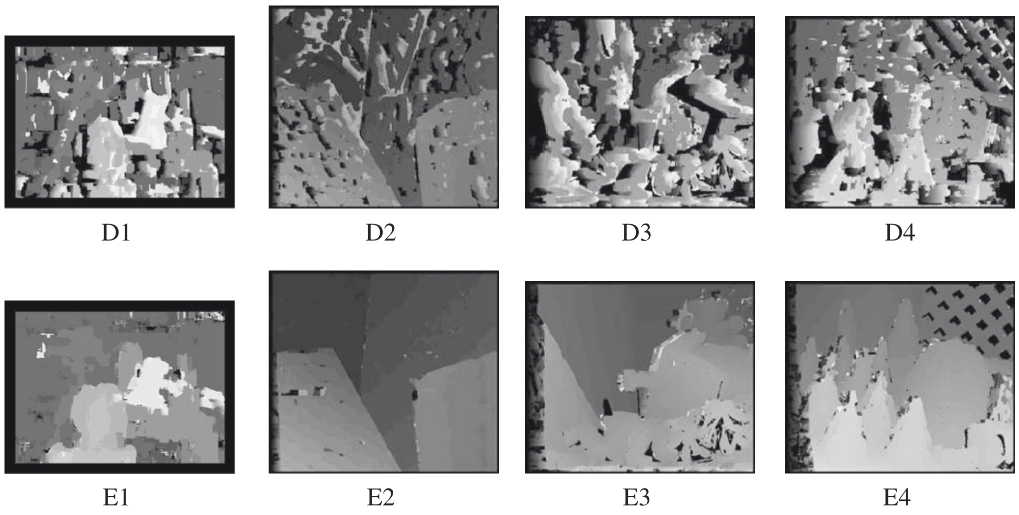


Fig. 12. Stereo correspondence results for Sum of Squared Difference (SSD) and Correlation (CORR) algorithms. Sub-images D1–D4 refer to the SSD algorithm while sub-images E1–E4 refer to the CORR algorithm.

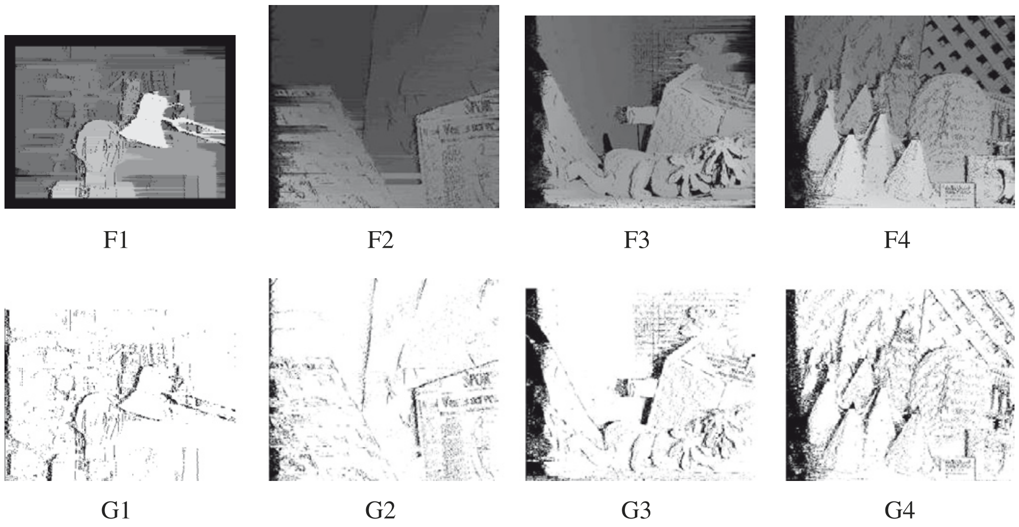


Fig. 13. Stereo correspondence results for the Dynamic Programming Maximum Likelihood Hardware implementation. Since both the DPML and DPMLHW results are visually indistinguishable, only the hardware results are shown here. Sub-images F1–F4 show the disparity estimates for the Tsukuba, Venus, Teddy and Cones data sets respectively, while sub-images G1–G4 show the occlusion estimates.

popular choices for other hardware stereo implementations. Table 1 and Figs. 11–13 provide a summary of these results.

4.2. Performance and timing

In this section the performance of a number of progressively better DPML implementations are compared. The label DPML refers to a C language software implementation of the dynamic programming stereo algorithm running on a 2.6 GHz Linux workstation. DPMLHW(S), DPMLHW(PP), DPMLHW(P) and DPMLHW(I) refer to FPGA hardware implementations. DPMLHW(S) is our first, largely sequential (row-wise processing), implementation, reported in [31]. DPMLHW(PP) is a partially pipelined hardware implementation that utilises the anti-diagonal structure of the cost and match matrix for improved performance and increased parallelisation. DPMLHW(P) is a fully pipelined hardware implementation that adds an extra pipeline buffer (PBUF) to split the large asynchronous cost computation into two distinct sections. This allows an increase in the clock frequency. Finally, DPMLHW(I) is a refinement of DPMLHW(P) in which the forward- and backward-passes of the algorithm are interleaved, and represents the final state of our design. Fig. 3 shows the DPMLHW(I) design's architecture.

Eq. (3) is the result of a *worst-case* timing analysis of the DPMLHW(I) hardware implementation, and gives the expected frame rate for an $n \times m$ image, maximum disparity D and FPGA clock rate F_{clk}:

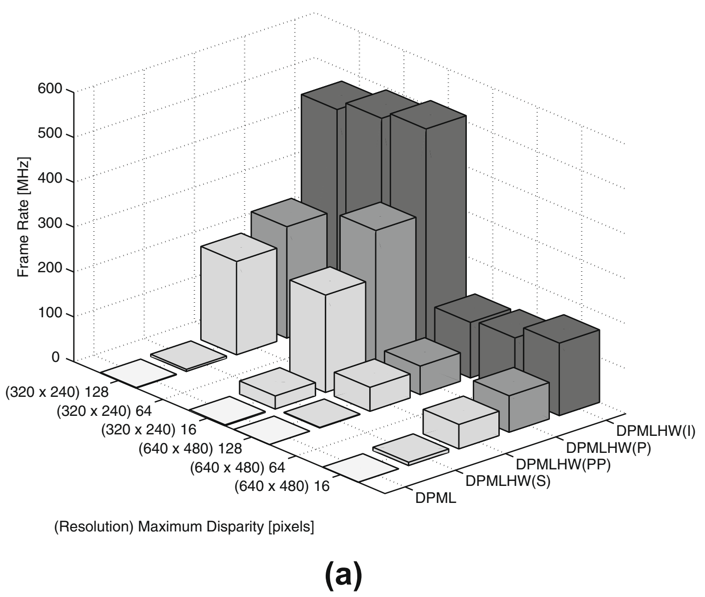
$$FPS_I(n,m,D,F_{clk}) = \left[\frac{2n + (2n + D/2 - 1)m}{F_{clk}}\right]^{-1}$$

where n and m indicates the width and height of the input image, D indicates the maximum disparity and F_{max_clk} the maximum allowable clock frequency of the circuit. In practice higher frame rates are typically seen, but this represents a lower bound on the processing frame rate. Similar timing equations have been developed for the other designs, but are not included here to avoid confusion.

A summary of runtime performance is shown in Table 2 for $D = 16$ and $D = 128$ (and $D = 64$ for the final implementation). The results are summarised graphically in Fig. 14. At a frame rate of 63.54 fps and pixel resolution of 640 × 480, the DPMLHW(P) FPGA implementation vastly out-performs equivalent software algorithms while maintaining comparably accurate results. This is a significant improvement over DPMLHW(S), which did not take advantage of the cost matrix structure. However, the best results are seen with the DPMLHW(I) implementation, running at 123.85 fps. A hardware implementation on the Xilinx Virtex 2 device ($D = 64$) achieved 131.14 fps. Due to BRAM limitations on the Xilinx Virtex 2 device, it was not possible to actually run the $D = 128$ DPMLHW(I) design in hardware. However, after performing post synthesis simulations targeted for the Xilinx Virtex 5 device it was noted that, in-fact, a frame rate of 123.85 fps can

Table 2
Frame rates achieved at image resolutions of 640 × 480 and 320 × 240 pixels. Highest frame rates are obtained from the pipelined and interleaved hardware, DPMLHW(I). The frame rates shown were calculated using formulae based on worst-case timing analysis, *i.e.* Eq. (3). Results for all DPMLHW designs, for 640 × 480 images, were verified an a Virtex 2 FPGA using the disparity ranges shown, with the exception of DPMLHW(I) with $D = 128$: this result is based on hardware simulation results with respect to a Virtex 5 device, as we did not have a Virtex 5 device to test with. Results for 320 × 240 images are not verified in hardware, but are included to provide direct comparison with authors who have used this image resolution.

Algorithm	D_{max}	F_{max_clk}	Resolution	FPS
DPMLHW(I)	128	80 MHz	640 × 480	123.85
	64	80 MHz	640 × 480	131.14
	16	100 MHz	640 × 480	161.54
	128	80 MHz	320 × 240	472.37
	64	80 MHz	320 × 240	494.80
	16	100 MHz	320 × 240	513.08
DPMLHW(P)	128	80 MHz	640 × 480	63.54
	16	100 MHz	640 × 480	81.16
	128	80 MHz	320 × 240	248.20
	16	100 MHz	320 × 240	323.75
DPMLHW(PP)	128	67 MHz	640 × 480	53.22
	16	67 MHz	640 × 480	54.37
	128	67 MHz	320 × 240	207.86
	16	67 MHz	320 × 240	216.91
DPMLHW(S)	128	47 MHz	640 × 480	1.15
	16	47 MHz	640 × 480	7.28
	128	47 MHz	320 × 240	4.60
	16	47 MHz	320 × 240	29.14
DPML	128	2.6 GHz	640 × 480	0.18
	16	2.6 GHz	640 × 480	0.24
	128	2.6 GHz	320 × 240	1.24
	16	2.6 GHz	320 × 240	2.01



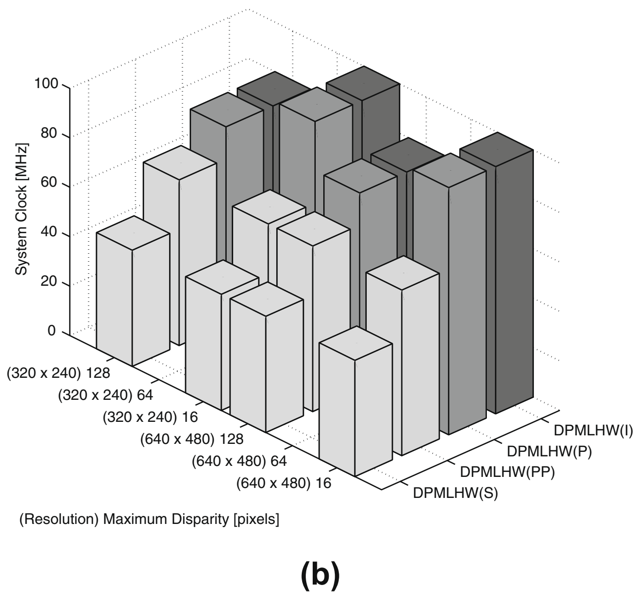


Fig. 14. The data of Table 2 in graphical format.

Table 3
A comparison of frame rates and depth pixels per second of various existing hardware implementations used to compute stereo correspondence. The prefix in front of the algorithm name represents the underlying technique: SAD for Sum of Squared Difference, CENSUS for census transform, PHASE for phase correlation and DPML for dynamic programming. The suffix indicates the authors associated with the implementation: MIYA – Miyajima et al. [16]; PERR – Perri et al. [18]; JACO: Jacobi et al. [36]; LEE – Lee et al. [22]; JIA – Jia et al. [37]; SIMH – Simhadri et al. [21] ; WOOD - Woodfill et al. [26]; MURPHY – Murphy et al. [27]; MITE – Mitéran et al. [19]; DIAZ – Diaz et al. [23]; MASR – Masrani et al. [25]; DARA – Darabiha et al. [24].

Algorithm	D_{max}	Resolution	F_{max_clk}	FPS	DPS	DPSN
DPMLHW(I)	128	640×480	80 MHz	123.85	4.870×10^9	60.87
DPMLHW(I)	64	640×480	80 MHz	132.87	2.612×10^9	32.65
DPMLHW(P)	128	640×480	80 MHz	63.54	2.477×10^9	30.96
SAD_MIYA	200	640×480	40 MHz	18.90	1.161×10^9	29.03
SAD_PERR	256	512×512	286 MHz	25.60	1.717×10^9	6.01
SAD_JACO	178	178×146	158 MHz	–	1.400×10^9	8.86
SAD_LEE	64	320×240	–	122.00	0.600×10^9	–
SAD_JIA	64	640×480	60 MHz	30.00	0.590×10^9	9.83
SAD_SIMH	64	512×512	100 MHz	0.34	0.006×10^9	0.06
CENSUS_WOOD	52	512×480	–	200.00	2.556×10^9	–
CENSUS_MURPHY	20	320×240	–	150.00	0.230×10^9 –	
PHASE_MITE	20	256×256	200 MHz	25.00	0.032×10^9	0.16
PHASE_DIAZ	4	640×480	65 MHz	211.00	0.259×10^9	3.98
PHASE_MASR	128	640×480	–	30.00	1.180×10^9	–
PHASE_DARA	20	256×360	–	33.00	0.061×10^9	–

be achieved, as predicted by Eq. (3), at a maximum disparity range of $D = 128$. Given the quality of the DPML results, we believe this represents the current state-of-the-art for hardware stereo disparity computation.

It is worth noting that it is possible to make further optimisations to DPMLHW(P) that do not involve interleaving the forward- and backward-passes, and these improvements result in theoretical performance[3] of roughly 100 fps for $D = 128$ on 640×480 images using a Virtex 5, although this design has not been tested on actual Virtex 5 hardware [28]. It runs at a clock speed of 125 MHz. The improvements rely in part on using the advanced features of the Virtex 5 family, such as *digital signal processing* (DSP) blocks. However, the greater improvements available via the DPMLHW(I) design have made this result obsolete.

Table 3 compares performance of our system with other hardware stereo systems in the literature. DPMLHW(I) shows the highest DPS (Disparity-Pixels/Second) measure, and with two implementations running in parallel it is capable of the highest

Table 4
Resource usage summary (pipelined + interleaved).

Pixels of disparity	Slices	4-input LUTs	BRAMs	MULT18 × 18s
16	2209 (5%)	3249 (3%)	72 (16%)	9 (2%)
32	3857 (8%)	6313 (7%)	136 (30%)	17 (3%)
64	7000 (15%)	10,576 (11%)	264 (59%)	33 (7%)
128	14,488 (32%)	23,027 (26%)	520 (117%)	65 (14%)

[3] Based on hardware simulation.

frame rate as well. Of the systems with frames rates at or above 150 fps, only Woodfill et al. has a DPS measure approaching that of DPMLHW(I). DPSN is DPS normalised by the system's clock rate, F_{max_clk}. It should be noted that timing performance of DPMLHW(I) can be further improved by duplicating the existing hardware module to process two or more scanlines simultaneously, at the expense of additional FPGA resources. For example, 640×480 images can be processed at well over 200 fps by instantiating two DPMLHW(I) cores in a suitable-sized FPGA device, and allowing them to process pairs of scanlines in parallel. In addition to the improved performance, note that our implementation is the only algorithm that attempts a globally (per-scanline) optimal solution for disparity.

4.3. FPGA resource usage

Despite the intensive nature of the dynamic programming problem, the DPMLHW(I) FPGA resource utilisation remains fairly low. On a Xilinx XC2VP100 device, at $D = 16$ this utilisation stands at: 2209 slices, or 5% of the FPGA resources. At $D = 128$, utilisation increases to 14,488 slices, or 32% of the resources. Note that the design also utilises special purpose DSP blocks for the RAM and multipliers. For $D = 16$ the utilisation of these blocks stands at: 16% and 2%, of total FPGA resources, respectively. At $D = 128$ this utilisation increases to: 117% and 14% respectively (we have run simulations of the $D = 128$ design as it exceeds available memory resources, but it will fit easily onto newer (existing) Xilinx FPGAs). Table 4 gives a complete resource usage summary.

5. Conclusions

This paper describes useful structure in the cost matrix of a large class of dynamic programming problems, namely that if any anti-diagonal in the cost matrix depends only on the preceding ones, then it is possible to compute cost matrix entries in parallel, in the time it takes the data to arrive. This allows for an efficient, pipelined architecture to be used in hardware implementations. An example design that implements stereo disparity estimation using dynamic programming is described. The interleaved and pipelined implementation runs at up to 131.14 fps for 640×480 images using a disparity range of $D = 64$ pixels, and in simulations runs at 123.85 fps for $D = 128$. For smaller images it is possible to perform disparity computations at frames rates well in excess of 200 fps. On a larger device, the processing rate can be doubled by instantiating two copies of the design in parallel with each other. The system has an accuracy essentially identical to the reference software implementation, and it fits on a moderately large FPGA device.

Acknowledgments

The authors would like to thank Ontario Centres of Excellence and MDA Space Missions for financial support, and MicroNet for providing the development platform. The authors would also like to thank Michael Greenspan (Queen's University), Lev Kirischian (Ryerson University) and Piotr Jasiobedzki (MDA Space Missions) for helpful discussions and suggestions related to this work.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press/McGraw-Hill, 2001.
[2] M. Borah, R. Bajwa, S. Hannenhalli, M. Irwin, A SIMD solution to the sequence comparison problem on the MGAP, in: Proceedings., International Conference on Application Specific Array Processors, 1994, pp. 336–345.
[3] D. Hoang, Searching genetic databases on splash 2, in: Proceedings. IEEE Workshop on FPGAs for Custom Computing Machines, 1993, pp. 185–191.
[4] W.S. Martins, J.B.D. Cuvillo, F.J. Useche, K.B. Theobald, G. Gao, A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison, in: In Pacific Symposium on Biocomputing 2001, 2001, pp. 311–322. <http://helix-web.stanford.edu/psb01/martins.pdf>.
[5] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, K. Tan, Generating parallel programs from the wavefront design pattern, in: IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium, IEEE Computer Society, Washington, DC, USA, 2002, p. 165.
[6] Ayala-Rincón, R. Jacobi, L. Carvalho, C. Llanos, R. Hartenstein, Modeling and prototyping dynamically reconfigurable systems for efficient computation of dynamic programming methods by rewriting-logic, in: Proceedings of the 17th Symposium on Integrated Circuits and System Design, SBCCI '04, ACM, New York, NY, 2004, pp. 248–253.
[7] T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 147 (1981) 195–197.
[8] M. Gong, R. Yang, L. Wang, M. Gong, A performance study on different cost aggregation approaches used in real-time stereo matching, International Journal of Computer Vision (IJCV) 75 (2) (2007) 283–296.
[9] S.M. Seitz, B. Curless, J. Diebel, D. Scharstein, R. Szeliski, A Comparison and evaluation of multi-view stereo reconstruction algorithms, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), vol. 1, 2006, pp. 519–526.
[10] B. Zitova, J. Flusser, Image registration methods: a survey, Image and Vision Computing 21 (2003) 977–1000.
[11] I.J. Cox, S.L. Hingorani, S.B. Rao, B.M. Maggs, A maximum likelihood stereo algorithm, Computer Vision and Image Understanding 63 (3) (1996) 542–567.
[12] D. Scharstein, R. Szeliski, A taxonomy and evaluation of dense two-frame stereo correspondence algorithms, International Journal of Computer Vision (IJCV) 47 (1–3) (2002) 7–42.
[13] M.Z. Brown, D. Burschka, G.D. Hager, Advances in computational stereo, IEEE Transactions on Pattern Analysis and Machine Intelligence 5 (8) (2003) 993–1008.
[14] M. Tappen, W. Freeman, Comparison of graph cuts with belief propagation for stereo, using identical MRF parameters, in: IEEE International Conference on Computer Vision (ICCV), vol. 2, 2003, pp. 900–906.
[15] Y. Lu, J.Z. Zhang, Q.M.J. Wu, Z.-N. Li, A survey of motion-parallax-based 3-D reconstruction algorithms, IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews 34 (4) (2004) 532–548.
[16] Y. Miyajima, T. Maruyama, A real-time stereo vision system with FPGA, in: International Conference on Field Programmable Logic and Applications (FPL), 2003, pp. 448–457.
[17] M. Hariyama, Y. Kobayashi, H. Sasaki, M. Kameyama, FPGA implementation of a stereo matching processor based on window-parallel-and-pixel-parallel architecture, in: Midwest Symposium on Circuits and Systems (MWSCAS), vol. 2, 2005, pp. 1219–1222.
[18] S. Perri, D. Colonna, P. Zicari, P. Corsonello, SAD-based stereo matching circuit for FPGAs, in: International Conference on Electronics, Circuits and Systems (ICECS), 2006, pp. 846–849.
[19] J. Miteran, J.-P. Zimmer, M. Paindavoine, J. Dubois, Real-time 3D face acquisition using reconfigurable hybrid architecture, EURASIP Journal on Image and Video Processing 2007 (1) (2007) 5.
[20] D. Han, D.-H. Hwang, A novel stereo matching method for wide disparity range detection, in: International Conference on Image Analysis and Recognition (ICIAR), 2005, pp. 643–650.
[21] V. Simhadri, P. Chandramani, Y. Ozturk, RASCor: Realtime Associative Stereo Correspondence, in: International Conference on Image Processing (ICIP), 2007.
[22] S. Lee, J. Yi, J. Kim, Real-time stereo vision on a reconfigurable system, Lecture Notes in Computer Science: Embedded Computer Systems: Architectures, Modeling, and Simulation 3553 (2005) 299–307.
[23] J. Diaz, E. Ros, S.P. Sabatini, F. Solari, S. Mota, A phase-based stereo vision system-on-a-chip, Biosystems 87 (2–3) (2006) 314–321.
[24] A. Darabiha, W.J. MacLean, J. Rose, Reconfigurable hardware implementation of a phase-correlation stereo algorithm, Machine Vision and Applications 17 (2) (2006) 116–132.
[25] D.K. Masrani, W.J. MacLean, A real-time large disparity range stereo-system using FPGAs, in: International Conference on Computer Vision Systems (ICVS), 2006.
[26] J.I. Woodfill, G. Gordon, R. Buck, Tyzx deepsea high speed stereo vision system, in: Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), vol. 3, 2004, pp. 41–46.
[27] C. Murphy, D. Lindquist, A. Rynning, T. Cecil, S. Leavitt, M. Chang, Low-cost stereo vision on an FPGA, in: Field-Programmable Custom Computing Machines, 2007, FCCM 2007, 15th Annual IEEE Symposium on, 2007, pp. 333–334.
[28] S. Sabihuddin, Dense stereo reconstruction in a field programmable gate array, Master's thesis, University of Toronto, 2008.
[29] J. Islam, Architecture and implementation of a high frame-rate stereo vision system, Master's thesis, Ryerson University, 2008.
[30] S. Sabihuddin, J. Islam, W.J. MacLean, Dynamic programming approach to high frame-rate stereo correspondence: a pipelined architecture implemented on a field programmable gate array, in: Canadian Conference on Electrical & Computer Engineering (CCECE), 2008, pp. 1461–1466.
[31] S. Sabihuddin, W.J. MacLean, FPGA implementation of dynamic programming stereo vision for high frame-rate tracking system, in: International Conference on Computer Vision Systems, Bielefeld, Germany, 2007.

[32] D. Buell, T. El-Ghazawi, K. Gaj, V. Kindratenko, High-performance reconfigurable computing, IEEE Computer 40 (3) (2007) 23–27.
[33] D. Scharstein, R. Szeliski, A taxonomy and evaluation of dense two-frame stereo correspondence algorithms, International Journal of Computer Vision 47 (1/2/3) (2002) 7–42.
[34] D. Scharstein, R. Szeliski, High accuracy stereo depth maps using structured light, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), vol. 1, 2003, pp. 195–202.
[35] D. Scharstein, R. Szeliski, 2007. <http://vision.middlebury.edu/stereo/>.
[36] R.P. Jacobi, R.B. Cardoso, G.A. Borges, Voc: a reconfigurable matrix for stereo vision processing, in: International Parallel and Distributed Processing Symposium (IPDPS), 2006.
[37] Y. Jia, X. Zhang, M. Li, L. An, A miniature stereo vision machine (MSVM-III) for dense disparity mapping, in: International Conference on Pattern Recognition (ICPR), 2004.