VLSI efficient discrete-time cellular neural network processor

M. Sindhwani, T. Srikanthan and K. Vijayan Asari

Abstract: Typical VLSI implementations of discrete-time cellular neural networks (DTCNN) incorporate costly hardware to implement the basic DTCNN cell, resulting in a small grid size that needs to be cascaded with many other chips for processing images of any practical size. In the paper, a low-cost DTCNN cell that can be incorporated into a single chip in large numbers has been proposed. Memory bandwidth considerations show that 256 DTCNN cells can be incorporated into a single chip DTCNN processor to compute a 256×256 image at 30 frames per second. Techniques based on rectangular-shaped cell grids for use with video memory have been proposed to satisfy the memory bandwidth requirements. The architecture of the proposed DTCNN processor is also capable of supporting the flexible grouping of basic cells. In addition, the processor, which is capable of supporting the flexible grouping of cells, can be cascaded in a highly scalable manner to facilitate the processing of larger images at high speed.

1 Introduction

Discrete-time cellular neural networks (DTCNN) [1] have received growing attention due to their ease of implementation in hardware. These networks can be implemented either in analogue hardware [2, 3] or in digital hardware [4–6]. A number of applications of the DTCNN have been presented in [1]. The use of the DTCNN for text segmentation and texture classification and segmentation has been proposed in [7]. In all these cases, the neighbourhood size is 1 or 2. Larger templates can also be decomposed [8] and it has been demonstrated that time-variant templates can be implemented due to the programmability of the architecture.

The DTCNN has several hardware implementations. Analogue implementations [2, 3, 9] lack accuracy due to device matching problems, and cannot be driven at high speeds [4]. Furthermore, it is difficult to connect neurochips and realise modifiable analogue synaptic weights [5].

Digital approaches using a digital neuron model recommend the use of a digital phase-locked loop (DPLL) [5] and a multi-input multilevel-quantised digital phase-locked loop (MM–DPLL) [10]. Although greater accuracy is achieved (all the internal components are digital), the system requires a phase modulated analogue input signal and the digital phase-locked loops have high hardware complexity, resulting in lower cell density [4].

The digital architecture proposed in [4] and [11] uses distributed arithmetic (DA). In this approach, a part of the DTCNN equation is precalculated on a host computer and downloaded to internal memory. The time-variant part is calculated using the DA function block. The internal resolution of the architecture is 11 bits and the 8-bit templates are programmable. However, real-time processing using this architecture is limited by its reliance on external circuitry. In addition, one word of storage is required for every pixel processed by the chip and the memory requirements explode to large values for real-world images.

CAM^2 [6] is a board-level solution, comprising a highly parallel array of DTCNN cells, an FPGA that controls the array, an RISC processor or DSP for serial data processing and some memory. The design has a high hardware cost and is a board-based solution. Also, the FPGA needs to be reprogrammed for every application.

In addition, their predecessors, cellular neural networks [12] (CNN) have also been implemented in analogue VLSI and on FPGA. A comparison is offered in [13]. On the basis of the tables in [4] and [13], Table 1 compares the various implementations. Most implementations do not cater for real-time applications, have a high hardware cost and lack support for processing large images. In addition, analogue approaches typically offer only 4-bit resolution. To overcome these shortcomings, a new digital implementation of the DTCNN is required.

One of the major challenges in the design of a digital DTCNN processor is to implement a low-cost DTCNN neuron that can be replicated to create larger networks. In addition, the proposed architecture must meet the bandwidth requirements associated with large images. This is particularly so if the neuron is small enough to be embedded into the IC in large numbers as all neurons must be kept fully occupied.

In this paper, we propose the design of a DTCNN processor that can be cascaded to support the real-time processing of large images. A low-cost digital DTCNN cell has also been designed to ensure that 256 DTCNN cells can be incorporated into a single chip. A number of such chips can be cascaded to form a highly integrated DTCNN processor.

IEE Proceedings online no. 20020322

DOI: 10.1049/ip-cds:20020322

Paper first received 21st May and in revised form 19th December 2001

M. Sindhwani and T. Srikanthan are with the Center for High Performance Embedded Systems, School of Computer Engineering, Nanyang Technological University, N4-B3b-05, Nanyang Avenue, Singapore 639798

K. Vijayan Asari is with the Department of Electrical & Computer Engineering, Old Dominion University, Norfolk, VA 23529, Virginia, USA

Table 1: Comparison of some current architectures

	Process	Area	Precision	No. of neurons	Frequency
CNN: analogue [13]	0.8 μm	30 mm² (~60 k gates eq.)	8-bit	32	1 MHz
CNN: FPGA [13]	—	160 k gates	8-bit	48	14 MHz
DTCNN: distributed arithmetic [4]	0.8 μm	389 μm × 463 μm	8-bit	9	30 MHz
DTCNN: OTA-based [3]	1.5 μm	290 μm × 275 μm	4-bit	1 cell	3.3 MHz

2 DTCNN model

A discrete-time cellular neural network consists of a grid of processing units called cells. Each unit is connected only to adjacent cells (neighbours). The cell on the ith row and jth column in a two-dimensional DTCNN is labelled as $C(i,j)$ or represented as μ. The r-neighbourhood N_r, of a cell $C(i,j)$ is defined by $N_r(i,j) = \{C(k,l)|i-k| <= r \ \& \ |j-k| <= r\}$. The following variables are defined for a cell μ:

(i) *Cell state*: this is defined as

$$x_\mu(n+1) = \sum_{\lambda \in N(\mu)} \boldsymbol{a}_{\lambda-\mu} y_\lambda(n) + \sum_{\lambda \in N(\mu)} \boldsymbol{b}_{\lambda-\mu} u_\lambda + \boldsymbol{I} \quad (1)$$

$\boldsymbol{a}$ and $\boldsymbol{b}$ are $(2r+1) \times (2r+1)$ matrices called feedback and control templates, $\boldsymbol{I}$ is the cell bias and, n is the iteration count.

(ii) *Cell output*: this is obtained from the cell state via the following equation:

$$y_\mu(n) = SGN(x_\mu(n)) \quad (2)$$

(iii) *Cell input*: u_λ represents external activation.

The next state, $x_\mu(n+1)$ does not depend on $x_\mu(n)$; this makes the hardware implementation of the DTCNN simpler. Also, local interconnections between cells and the translation invariance of the templates mean that the basic cells are regular and identical.

A block diagram of the DTCNN neuron is presented in Fig. 1. As shown in (1), the cell state consists of two parts:

(i) The part that stays constant while processing any image (written as $\boldsymbol{\Sigma Bu + I}$).

(ii) The part that varies with every iteration (written as $\boldsymbol{\Sigma Ay(n)}$).

The part that varies with every iteration depends on the output of the neighbouring cell in the previous iteration. According to (2), this will be $+1$ or -1. Thus, the variable part of (1), will become

$$\begin{aligned} \boldsymbol{a}_{\lambda-\mu} y_\lambda(n) &= \boldsymbol{a}_{\lambda-\mu} \quad \text{if } y_\lambda(n) = +1 \\ &= -\boldsymbol{a}_{\lambda-\mu} \quad \text{if } y_\lambda(n) = -1 \end{aligned} \quad (3)$$

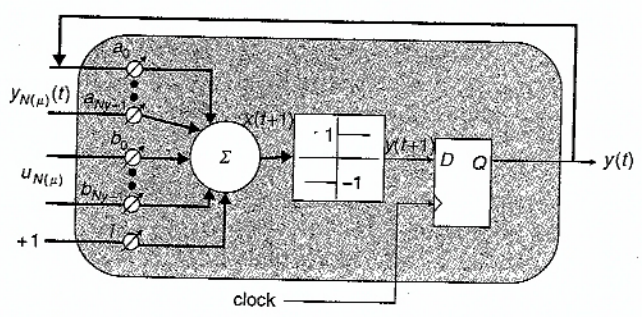

Fig. 1 *Block diagram of a DTCNN neuron*

Clearly, the variable part of (1) can be implemented as a series of additions or subtractions (no multiplication necessary) and the frame-constant part can be precalculated.

3 Design overview

In this Section, some of the design specific considerations of the DTCNN processor are discussed, along with some of the applicable solutions.

3.1 Processing large images

Since the DTCNN is a one-processor-per-pixel network, a 256×256 image requires 65536 DTCNN cells. Since it is not practicable to incorporate such a large number of DTCNN cells on a single chip, alternate algorithms for processing large images must be introduced. These include time multiplexing the DTCNN Cell Grid (divide the image into blocks and process it one block at a time), cascading multiple chips (divide the image into blocks and process each block on a separate DTCNN chip) or a combination of both approaches. Table 2 compares the requirements for processing a 1024×1024 image in the various configurations, for a DTCNN chip containing 256 neurons (block size = 256 pixels). As can be seen, processing more than 256 blocks per chip (65536 pixels per chip) results in very high on-chip RAM requirements and takes significantly longer to process. However, cascading multiple chips makes it possible to better cope with such concerns.

3.2 Memory bandwidth

A cell-based approach allows designers to embed a large number of DTCNN cells into a single chip to achieve high performance. However, integrating a large number of

Table 2: Requirements to process a 1024 × 1024 image on 256-neuron DTCNN chips†

Configuration	Blocks processed	No. of chips	Time	On-chip RAM#
Cascading	1	4096	1x*	0 units
Combined	16	256	16x	4096 units
Combined	32	128	32x	8192 units
Combined	64	64	128x	16384 units
Combined	256	16	256x	65536 units
Combined	1024	4	1024x	262144 units
Combined	2048	2	2048x	524288 units
Time multiplexed	4096	1	4096x	1048576 units

† Each chip has 256 DTCNN neurons
* x is the time taken to complete one iteration
# RAM requirements are given for each chip (can be shifted off-chip)

DTCNN cells onto a single chip necessitates large data paths between the input image and the DTCNN cell grid.

There are three possible ways to tackle the issue of memory bandwidth. The simplest and most effective way is to buffer the image. We could choose to store the entire input image on the chip itself. This is an expensive option that may only be considered for very time-critical cases. A more practical approach would be to use a larger data bus. By using a wider bus, more data can be read into the system at one time. A 32-bit bus relaxes the requirements by a factor of four, against an 8-bit bus. This might be a preferred option, if sufficient number of port lines could be made available for memory access. Of the various memory buffering techniques that can be employed, we have chosen to model our architecture to benefit from the high-speed serial ports of video memory, which is discussed next.

3.2.1 Video memory: Video memory modules are attractive as the data corresponding to an entire row of pixels can be moved into a shift register in a single access. The shift register can then be clocked at very high rates to provide data serially on a single data line. Since video RAM is dual-ported, it serves well to satisfy the bandwidth requirements of the DTCNN processor provided the data to all on-chip neurons can be loaded serially.

Since the entire row of data is available, the processing elements should ideally be arranged in a single row so that all the data coming out of the shift register can be catered for simultaneously. Although it is typical to expect the DTCNN cell grid to be square, we incorporated rectangular grids to benefit from the video RAM. It has been verified that the DTCNN works equally well in rectangular grids, such as 1×256. The data is clocked out of the shift register and passes into the DTCNN chip on a single input line as shown in Fig. 2. Internally, the data passes to each of the processing elements. Processing can start as soon as the data corresponding to the neighbourhood has been loaded.

This arrangement requires more on-chip memory since the row of processing elements needs all the data within its two-neighbourhood. The neighbourhood of a 1×256 DTCNN grid is shown in Fig. 3. This necessitates the storage of five rows of data consisting of 1280 pixels (256×5) on-chip.

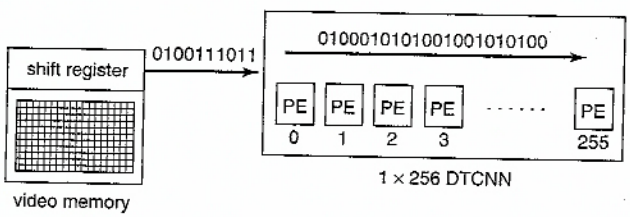

Fig. 2 *1 × 256 DTCNN grid connected to video memory*

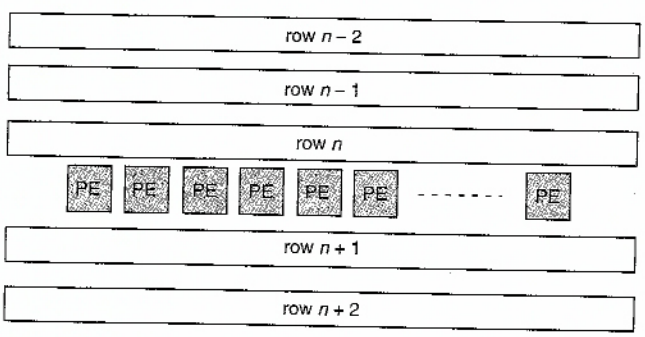

Fig. 3 *Two-neighbourhood for a 1 × 256 DTCNN grid*

In this approach to computation, increase in on-chip memory does not affect the data rate. This is because after the first four rows have been loaded, only one row of data (i.e. 256 pixels) needs to be fetched at a time. After a particular row has been processed, 'row $n-2$' can be overwritten. The other four rows of data can be shifted up and only 'row $n+2$' needs to be loaded. This is easily implemented in hardware by using a counter-based approach that creates a circular queue to simulate the shifting.

4 DTCNN processor

Fig. 4 shows a flexible and scalable DTCNN processor. The architecture is designed to handle 8-bit grayscale inputs with 12-bit templates. The neighbourhood size is fixed at two and the processor contains an $M \times N$ grid of processing elements. The regular and identical structure of the processing elements permits the use of a global control unit (GCU) that generates the necessary control signals. Each unit is directly connected to its closest two neighbours in all directions. The memory interface-1 unit obtains the image from the input as a series of $M \times N$ blocks. The templates are stored in the template RAM. Since the weights are stored in RAM, they can be overwritten to make the system programmable to facilitate reconfigurability. The programming unit allows the programming of parameters such as the templates, the number of iterations and the number of blocks that the image is divided into. The outputs are transferred to the outside world by the memory interface-2 unit. The cascade controller interfaces to other DTCNN chips cascaded with this one.

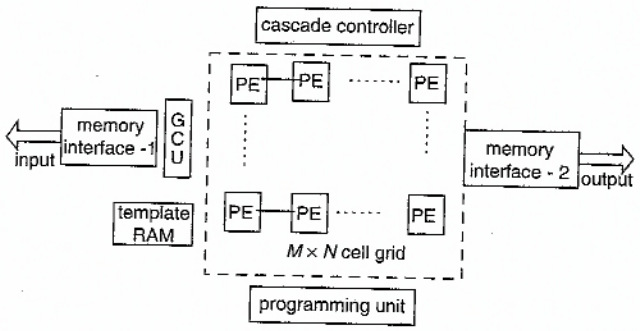

Fig. 4 *DTCNN processor*

Large images are processed by time multiplexing the DTCNN cell grid. The image is divided into a set of blocks (of size $M \times N$), each block is loaded onto the cell grid and processed, the results stored and then the next block is loaded. At each iteration, there are two parts to the context of the block, namely the constant part and the result part. The first part requires one word of data for every single pixel that is processed. This explodes to a very large memory requirement while the second part only requires one bit of storage for every pixel processed.

To avoid the large memory requirements of the first part, we have opted to recompute the constant part for every block at every iteration. This requires more processing time but results in significant hardware savings, thereby reducing the memory requirements essentially to one bit per pixel processed. Although this results in a higher memory bandwidth requirement, it is easily met by using modern video memory.

The architecture also has a cascade controller that interfaces to other DTCNN chips to be cascaded. This allows a large image to be divided into a number of macroblocks, each of which is processed on one chip. Neighbourhood information is propagated to the connected chips by the cascade controller and each macroblock may be processed by time multiplexing the DTCNN cell grid.

The optimal size of the DTCNN cell grid is influenced by a number of factors; memory data bandwidth requirements, processing time, frame rate, on-chip memory and the size of the image to be processed. Based on our analysis, it was decided that 256 DTCNN neurons are embedded onto a single chip. A 256 × 256 image can be processed in real-time (30 frames per second) by time multiplexing the DTCNN cell grid. In addition, it can be connected to other DTCNN processor chips through the cascade controller for processing large images (as shown in Table 2). The increased bandwidth requirement is satisfied through the use of appropriate video memory. The desired properties and basic timing constraints for the design of the DTCNN Processor for Image Processing are shown in Tables 3 and 4.

Table 3: Constraints for the DTCNN processor design

Frame rate	30 frames/second [real-time]
Frame size	256 × 256
Maximum iterations to convergence	100
Input image	8-bit grey-scale
Template precision	12 bits
Neighbourhood depth	2
Output image	binary (1-bit)

Table 4: Timing constraints with a 256-neuron cell grid

To process 1 frame	33.333 ms
Time per iteration	333.33 μs
Number of blocks	256
Processing time per block-iteration	1.302 μs

4.1 Implementation of the DTCNN cell in VLSI

The cell-state equation can be divided into a constant part and a variable part (as discussed in Section 2). A multiply–accumulate unit (MAC) is required for calculating the constant part. The variable part of (1) can be implemented as a series of additions and subtractions.

The processing for the neuron is divided into three steps:

Step 1. Multiplication of the control template element ['*b*'] by the input pixel value ['*u*'] (This is a 12-bit by 8-bit multiplication and requires a 12-bit adder). The result is a 20-bit product, which is too high a resolution for a system in which the final value depends on only the sign of the final accumulation. Thus, only the more significant 12 bits are used for further steps.

Step 2. Depending on the sign of the output from the corresponding neighbour, the feedback template element ['*a*'] is added or subtracted from the truncated product obtained in step 1. To avoid overflow, it is advisable to use a 13-bit adder at this stage.

Step 3. This is followed by the accumulate stage. In this stage, the result obtained in step 2 is added to the value stored in the accumulator.

The algorithm for the processing carried out by the neuron is shown in Fig. 5 and the architecture of the DTCNN neuron is shown in Fig. 6.

I. Upon RESET, load bias into Accumulator (ACC ← bias)
II. FOR the desired number of iterations (maximum 100)
  A. FOR all input and output cells in the neighborhood
    1. Multiply input by template (temp1 ← $u_{\lambda-\mu} \times b_{\lambda-\mu}$)
    2. Add result to Accumulator (ACC ← ACC + temp1)
    3. Multiply output by template (temp2 ← $y_{\lambda-\mu}(n-1) \times a_{\lambda-\mu}$)
    4. Add result to Accumulator (ACC ← ACC + temp2)
  B. END FOR
III. END FOR
IV. Output is the sign of the Accumulator ($y_{\mu}(n)$ = ACC.MSB)
V. STOP.

Fig. 5 *Algorithm for processing by the DTCNN neuron*

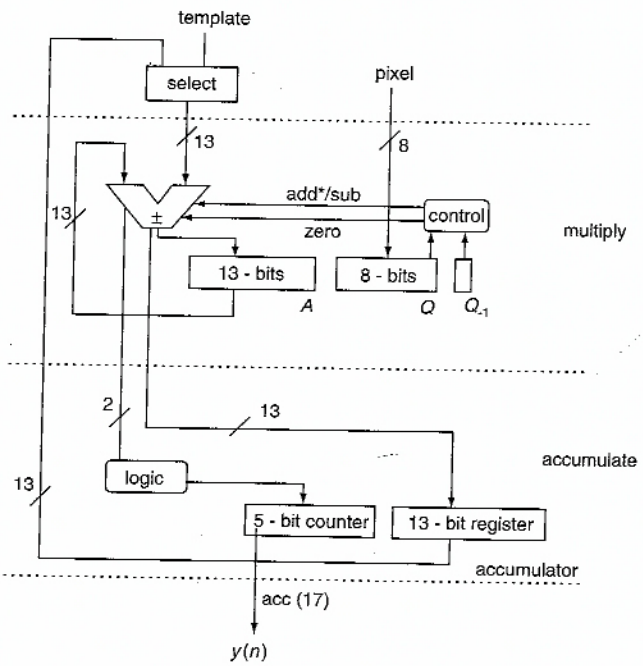

Fig. 6 *DTCNN neuron*

Since all steps require an adder/subtractor circuit, resource utilization can be maximised by reusing the same adder. The multiplication, performed using Booth's algorithm, takes eight cycles for a 12-bit by 8-bit multiplication. In the next cycle, the '*a*' template value is added to or subtracted from the 12-bit result obtained in the previous step. Since the addition is of two 12-bit values, the adder width is maintained at 13 bits to avoid overflow.

This is followed by the accumulate step. This is the addition of an 18-bit value stored in the accumulator with a 13-bit value produced at the end of step *b*. For this addition, the 13-bit value needs to be sign-extended. Sign extension requires the replication of the MSB of the 13-bit value to the remaining five most significant bits.

Since the multiplication circuit already consists of a 13-bit adder, it is reutilised to add the lower 13 bits at this

step. The upper five bits are added separately. The operation of this five-bit adder depends on the carry produced in the addition of the lower 13 bits and the sign of the result produced in step 2. The possible operations are summarised in Table 5. Consequently, the five-bit adder was implemented as an up/down counter with 'zero' control. Combinational logic decides whether the five-bit counter increments by one decrements by one, or simply retains the old value.

Table 5: Operation of the five-bit adder

Carry[#]	MSB[†]	Sign extended value	Decimal	Operation	Result
0	0	00000B	0	Acc = Acc + 0 + 0	Acc
0	1	11111B	−1	Acc = Acc + 0 − 1	Acc − 1
1	0	00000B	0	Acc = Acc + 1 + 0	Acc + 1
1	1	11111B	−1	Acc = Acc + 1 − 1	Acc

[#]Carry is the carry generated from the addition of the lower 13-bits
[†]MSB is the most significant bit of the value produced in step *b* of the algorithm

The hardware neuron requires a 13-bit adder, combinational logic for control, a 5-bit up/down counter and 35 bits of memory. In addition, a 1-bit register is required to remember the previous output.

4.2 13-bit hybrid adder

It was found that implementing a 13-bit ripple carry adder (RCA) was producing the result too close to the required deadline and a 13-bit carry-look-ahead adder (CLA) had a significantly higher hardware cost, but provided no gains since our deadlines were met with large margins by the CLA implementation. This led to the hybrid design in which the combination of an eight-bit tipple carry adder with a five-bit carry-look-ahead adder was used.

4.3 Area–time measures

The design was simulated for functional verification in Synopsys VSS version 2000.02 and area parameters were obtained after synthesis, place and route of the architecture in the Synopsys Design Analyzer version 1999.10 and Apollo 2000.2.3. In 0.35 μm VLSI technology, the DTCNN neuron requires a total cell area (including interconnect) of just under 1700 units. The critical path consists of adders. Each block-iteration requires 250 cycles. At 200 MHz, a single iteration takes 1.25 microsecond, which is less than the timing constraint of 1.302 microseconds. With a 1×256 or 16×16 cell grid at 200 MHz, a 256×256 image can be processed in real time (30 frames per second).

5 Conclusions

The local interconnection, translation invariant templates, inherent parallelism and ease of implementation of the system equation make the DTCNN extremely attractive for VLSI implementation. A low-cost DTCNN cell, with a cell area of 1700 units, has been implemented and tested using Synopsys Design Compiler 1999.10. It has been placed and routed using Apollo 2000.2.3 and the area–time analyses show that 256 DTCNN cells can be easily incorporated into a single IC. The DTCNN cell grid for the proposed processor has been organised as a rectangular grid to benefit from the high-speed serial port of VRAMs. It has been shown that a single processor is capable of processing a 256×256 image at 30 frames per second. Finally, the proposed DTCNN processor can be dynamically reconfigured to support a variety of image processing tasks as well as readily cascaded to cater for large images and different neighbourhood sizes.

6 References

1 HARRER, H., and NOSSEK, J.A.: 'Discrete-time cellular neural networks', *Int. J. Circuit Theory Appl.*, 1992, **20**, pp. 453–467
2 RODRIGUEZ-VAZQUEZ, A., ESPEJO, S., DOMINGUEZ-CASTRO, R., HUERTAR, J.L., and SANCHEZ-SINENCIO, E.: 'Current mode techniques for the implementation of continuous- and discrete-time cellular neural networks', *IEEE Trans. Circuits Syst.*, 1993, **CAS-40**, pp. 132–146
3 HARRER, H., NOSSEK, J.A., and STELZL, R.: 'An analog implementation of discrete- time cellular neural networks', *IEEE Trans. Neural Netw.*, 1992, **20**, pp. 466–476
4 PARK, S., LIM, J., and CHAE, S.: 'Digital implementation of discrete-time cellular neural networks with distributed arithmetic', Proceedings of IEEE Int. Conf. on Neural networks, Houston, USA, 1997, Vol. 2, pp. 959–963
5 UEDA, T., TAKAHASHI, K., HO, C.-Y., and MORI, S.: 'Discrete-time cellular neural networks using digital neuron model with DPLL', Proceedings of the Second International Workshop on Cellular neural networks and their applications, 1992, pp. 252–257
6 IKENAGA, T., and OGURA, T.: 'A DTCNN universal machine based on highly parallel 2-D cellular automata CAM^{2}', *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, 1998, **45**, pp. 538–546
7 KELLNER, A., MAGNUSSEN, H., and NOSSEK, J.A.: 'Texture classification, texture segmentation and text segmentation with discrete-time cellular neural networks', Proceedings of the Third International Workshop on Cellular neural networks and their applications, 1994, pp. 243–248
8 MIRZAI, B., LIM, D., and MOSCHYTZ, G.S.: 'Applications of CNN processing by template decomposition'. Proceedings of the 5th IEEE International Workshop on Cellular neural networks and their applications, 1998, pp. 379–384
9 HARRER, H., and NOSSEK, J.A.: 'Discrete-time cellular neural networks', Proceedings of the Second International Workshop on Cellular neural networks and their applications, 1992, pp. 163–168
10 TOKUNAGA, M., and MORI, S.: 'Digital neuron model with DPLL for associative memory', *IEEE Int. Symp. Circuits Syst. Proc.*, 1990, **2**, pp. 1069–1072
11 PARK, S., LIM, J., and CHAE, S.: 'Discrete-time cellular neural networks using distributed arithmetic', *Electron. Lett.*, 1995, **31**, pp. 1851–1852
12 CHUA, L.O., and YANG, L.: 'Cellular neural networks: Theory', *IEEE Trans. Circuits Syst.* 1988, **35**, pp. 1257–1272
13 WIEHLER, K., PEREZOWSKY, M., and GRIGAT, R.: 'A detailed analysis of different CNN implementations for a real-time image processing system', Proceedings of 6th IEEE International Workshop on Cellular neural networks and their applications (CNNA 2000), 2000, pp. 351–356