R.T.
YILDIZ TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

DESIGN OF A CELLULAR NEURAL NETWORK EMULATOR AND ITS IMPLEMENTATION ON AN FPGA DEVICE

NERHUN YILDIZ

Ph.D. THESIS
DEPARTMENT OF ELECTRONICS AND COMMUNICATIONS
ENGINEERING
PROGRAM OF ELECTRONICS

SUPERVISOR
PROF. DR. VEDAT TAVŞANOGLU

İSTANBUL, 2013

R.T.

YILDIZ TECHNICAL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

DESIGN OF A CELLULAR NEURAL NETWORK EMULATOR AND ITS IMPLEMENTATION ON AN FPGA DEVICE

A thesis submitted by Nerhun YILDIZ in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY** is approved by the committee on December 21, 2012 in Electronics and Communications Engineering Department, Electronics Programme.

Supervisor
Prof. Dr. Vedat TAVŞANOĞLU
Yıldız Technical University

Examining Committee Members
Prof. Dr. Vedat TAVŞANOĞLU
Yıldız Technical University ____________________

Assoc. Prof. Dr. Müştak Erhan YALÇIN
İstanbul Technical University ____________________

Asst. Prof. Dr. Ertuğrul SAATÇI
İstanbul Kültür University ____________________

Prof. Dr. Oruç BİLGİÇ
İstanbul Kültür University ____________________

Asst. Prof. Dr. Burcu ERKMEN
Yıldız Technical University ____________________

This study was supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under project number 108E023.

ACKNOWLEDGMENTS

First of all, writing this thesis is more than just another step in my academic carrier, it was one of my childhood dreams. I always envied the title, Dr., when I was watching a movie or reading a book, not to mention attaching too much meaning to it. But still, it feels good and right to get the title, anyway.

I want to thank to all the individuals in my life that make the preparation of this thesis possible. First, I want to thank Endam, my wonderful wife, for her support and patience; for this thesis will not exist without her. If truth be told, I honestly don’t know from whom shall I continue; but second, I want to thank my professor and instructor Dr. Vedat Tavşanoğlu for his guidance and contribution in my life, academic or otherwise. Third, I want to thank Evren Cesur for both his academic contribution to my thesis, and sharing my assistance workload at the university at times when I needed the most. Fourth, I want to thank Murathan Alpay for again helping me to make time for my thesis, and continue with Dr. Umut–Engin Ayten, Işıl Kalafat, Oğuzhan Yavuz, Nergis Tural–Polat, Tankut Açar and all my other colleagues who had given me support. Then, I want to thank all my instructors who shared their knowledge with me and guided me, from my primary school teachers to the professors of my PhD courses. Last but not least, I want to thank my family, for raising me to be the person I am now. Thank you all.

January, 2013

Nerhun Yıldız

CONTENTS

Page

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

CHAPTER 6

LIST OF SYMBOLS

p	number of dimensions of a CNN
q	number of layers of a CNN
m	neighborhood of the spatial interconnactions
I	number of cells in the first spatial dimension of a 2–D CNN
J	number of cells in the second spatial dimension of a 2–D CNN
i	space variable of the first spatial dimension of a 2–D CNN
j	space variable of the second spatial dimension of a 2–D CNN
$C(i,j)$	CNN cell
t	time variable
$x_{ij}(t)$	cell state of a $C(i,j)$ cell at time t
$\dot{x}_{ij}(t)$	time derivative of $x_{ij}(t)$
$y_{ij}(t)$	output of a $C(i,j)$ cell at time t (output pixel in case of an image)
u_{ij}	constant–valued input of a $C(i,j)$ cell (input pixel in case of an image)
k	space variable of the first spatial dimension of a CNN template
l	space variable of the second spatial dimension of a CNN template
a_{kl}	constant–valued feedback coefficients
b_{kl}	constant–valued input coefficients
z	threshold value
$f(.)$	output function
$\boldsymbol{A}$	feedback template
$\boldsymbol{B}$	input template
$\circledast$	template–dot–product operator
$\boldsymbol{Y}_{ij}(t)$	translated masked output image
$\boldsymbol{U}_{ij}$	translated masked input image
n	discrete–time variable
$x_{ij}(n)$	discrete–time state of a $C(i,j)$ cell
$y_{ij}(n)$	discrete–time output of a $C(i,j)$ cell
$\boldsymbol{Y}_{ij}(n)$	translated masked output image of discrete–time CNN
T_s	sampling period
$\bar{\boldsymbol{A}}$	discrete–time $\boldsymbol{A}$ template
$\bar{\boldsymbol{B}}$	discrete–time $\boldsymbol{B}$ template
$\bar{a}_{kl}$	discrete–time constant–valued feedback coefficients
$\bar{b}_{kl}$	discrete–time constant–valued input coefficients
$\bar{z}$	discrete–time threshold value
$\mathbf{U}$	input matrix
$\mathbf{Y}$	output matrix
N	number of iterations

K	number of iterations unrolled
L	number of vertical stripes
g_{ij}	intermediate constant of a $C(i,j)$ cell
S	state inputs of a CASTLE/Falcon processor
T	template coefficient inputs of a CASTLE/Falcon processor
$\mathbf{G}$	intermediate constant value matrix

LIST OF ABBREVIATIONS

1–D	One–Dimensional
2–D	Two–Dimensional
ADC	Analog to Digital Converter
APU	A Processing Unit
ASIC	Application Specific Integrated Circuits
BPU	B Processing Unit
BRAM	Block RAM
BW	Black and White
CNN	Cellular Neural Network
CNN–UM	CNN Universal Machine
CT CNN	Continuous–Time CNN
DiROM	Distributed Read Only Memory
DSP	Digital Signal Processors
DT CNN	Discrete–Time CNN
DVI	Digital Visual Interface
FPGA	Field–Programmable Gate Array
FSR	Full Signal Range
GPU	Graphical Processing Unit
HDMI	High–Definition Multimedia Interface
I/O	Input/Output
ID	Identity Document
LVDS	Low–Voltage Differential Signaling
MAC	Multiply and ACcumulate operations
MSBs	Most Significant Bits
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PLL	Phase–Locked Loop
RAM	Random Access Memory
RGB	Red Green Blue
RS232	Recommended Standarts 232
RTCNNP	Real–Time CNN Processor
SFR	Special Function Register
TÜBİTAK	The Scientific and Technological Research Council of Turkey
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VHDL	Very–high–speed integrated-circuit (VHSIC) Hardware Description

	Language
xPU	x Processing Unit

LIST OF FIGURES

Page

LIST OF TABLES

ABSTRACT

DESIGN OF A CELLULAR NEURAL NETWORK EMULATOR AND ITS IMPLEMENTATION ON AN FPGA DEVICE

Nerhun YILDIZ

Department of Elektronics and Communication Engineering

Ph.D. Thesis

Supervisor: Prof. Dr. Vedat TAVŞANOĞLU

It is well known that technology affect our everyday lives and change them significantly from the beginning of humanity. As the technology grows more rapidly in the last few decades, the changes also started to occur more frequently. For example, a few centuries ago, a person could experience at most one significant leap of change in his or her life; but today, a senior may have experienced the leaps caused by the inventions of the television, transistors, satellites, computers, cellular phones, other portable electronics, etc.

The rapid change of the technology also create trends of new research topics, like image processing, which was nothing more than a television or camera engineers or academics specialty just 20 years ago. Furthermore, the processing was limited by preserving, transmitting and receiving images with minimum noise and distortion. With the introduction of digital cameras, countless new ideas of image processing emerged, e.g., image enhancement, image compression, automated target recognition and tracking, biometric recognition, etc. There are two main difficulties in the application of these ideas: (1) new image processing algorithms should be developed and implemented within tight time frames and (2) fast and parallel processors are required to match the computation intensity of the real–time image processing.

On the other hand, a Cellular Neural Network (CNN) is a multi–dimensional signal processing paradigm, whose analog and digital 2–D implementations can be used in image processing. The main advantage of any CNN implementation is that, many image processing algorithms can be implemented on the same structure, solving the first problem mentioned above. On the other hand, analog CNN implementations are known to operate at speeds up to 10 kilo–frames/s for grayscale images with resolutions lower then 176×144, which seems to solve the second problem. However, this is not the case for high–resolution and medium frame–rate images like full–HD 1080p@60 (1920 × 1080 resolution, 60 Hz frame rate), where the performance of the analog implementations drop below the real–time limits. Then again, the digital implementations of CNN does not have the intrinsic parallel connectivity of their analog counterparts, consequently, none of the digital CNN implementations are reported to operate for full–HD 1080p@60.

In this thesis, an improved real–time digital CNN architecture capable of processing full–HD 1080p@60 video images is proposed, described in VHDL and realized on two different FPGA devices. The architecture is designed to have superior properties over its predecessors. First, the architecture is highly scalable, which is proven by implementing the same design on a high–end and a low–cost FPGA device. Second, most parts of the structure are designed to be reconfigurable and flexible, e.g., the size of the CNN templates, fixed–point bit–widths of all signals, the number of iterations, etc. Third, most parameters like template coefficients, bias, boundary conditions and bypass modes are programmable at runtime. The architecture proposed in this thesis is the only CNN implementation reported in the literature that assemble all of these features together.

Keywords: cellular neural networks, image processing, field–programmable gate-arrays, real–time systems

---

YILDIZ TECHNICAL UNIVERSITY

GRADUATE SCHOOL OF NATRAL AND APPLIED SCIENCES

ÖZET

BİR HÜCRESEL SİNİR AĞI EMÜLATÖRÜNÜN TASARLANMASI VE FPGA ÜZERİNDE GERÇEKLENMESİ

Nerhun YILDIZ

Elektronik ve Haberleşme Mühendisliği Anabilim Dalı

Doktora Tezi

Tez danışmanı: Prof. Dr. Vedat TAVŞANOĞLU

İnsanlığın başından itibaren günlük hayatımızı etkileyen ve değiştiren en önemli etkenlerden birinin teknoloji olduğu şüphesiz bir gerçektir. Teknolojideki gelişmenin son birkaç on yıl içinde iyice hızlanmasıyla bu değişimlerin sıklığı da artmıştır. Örneğin birkaç yüzyıl önce yaşamış bir insanın hayatı boyunca gözlemleyebileceği değişim sayısı en fazla bir iken, günümüzde yaşayan yaşı ilerlemiş bir bireyin hayatı televizyon, transistör, uydu, bilgisayar, cep telefonu ve diğer taşınabilir elektronik cihazlar gibi teknolojik gelişimler ile defalarca etkilenmiştir.

Teknolojideki bu hızlı gelişim aynı zamanda araştırma konularında da yeni eğilimlerin ortaya çıkmasına neden olmaktadır. Eğilimin arttığı bu konulardan biri de görüntü işlemedir. Bundan 20 yıl öncesine kadar uzmanlığı görüntü işleme olan kişiler yalnızca televizyon ve video kamera tasarım mühendisleri ile konuyla ilgilenen akademisyenlerdi. Ayrıca dönemin görüntü işleme konularının neredeyse tamamı görüntünün kalite kaybı veya bozulma olmadan saklanması ve iletilmesi ile sınırlıydı. Sayısal kameraların ortaya çıkıp yaygınlaşmasıyla beraber görüntü iyileştirmeden görüntü sıkıştırmaya, otomatik hedef takibi ve tanımadan biyometrik tanıma sistemlerine kadar birçok yeni görüntü işleme fikri ortaya çıkmaya başladı. Ancak bu fikirlerin hayata geçirilmesinde iki temel problem ortaya çıktı: (1) Yeni algoritmaların sınırlı zamanda geliştirilmesi ve sistem olarak gerçeklenmesi ile (2) hesaplamaların gerçek zamanlı olarak yapılabilmesi için hızlı ve paralel işlem yapma yeteneği olan donanımların gerekmesi.

Öte yandan Hücresel Sinir Ağları (Cellular Neural Networks – CNN) çok boyutlu ortamlar üzerinde işlem yapma yeteneği olan bir yapı olarak ortaya atılmıştır ve iki boyutlu analog ve sayısal gerçeklemeleri görüntü işlemede kullanılabilmektedir. Herhangi bir CNN gerçeklemesinin en büyük avantajı, aynı yapı üzerinde birçok farklı algoritmanın gerçeklenebilmesi sayesinde yukarıda bahsedilen ilk probleme çözüm oluşturmasıdır. Ayrıca analog CNN gerçeklemelerinin 176 × 144 veya daha düşük çözünürlükteki gri seviyeli görüntüler için 10 kilo çerçeve/s işlem hızına ulaşabilmesi dolayısıyla ikinci problemin çözümüne de aday olduğu bir gerçektir. Ancak full–HD 1080p@60 (1920 × 1080 çözünürlük, 60 Hz çerçeve hızı) gibi yüksek çözünürlüğe ve orta seviyede çerçeve hızına sahip görüntüler söz konusu olduğunda analog yapıların hızı gerçek zamanlı gerçekleme sınırının altına düşmektedir. Sayısal CNN gerçeklemeleri ise analog yapılardaki doğal paralel hesap özelliğine sahip olmadıklarından dolayı full–HD 1080p@60 için çalışan bir gerçekleme literatürde yer almamaktadır.

Bu tezde full–HD 1080p@60 video görüntülerini işleyebilen gelişmiş bir gerçek zamanlı sayısal CNN mimarisi önerilmiş, VHDL dilinde kodlanmış ve iki farklı FPGA üzerinde gerçeklenmiştir. Tasarlanan mimarinin önceki tasarımlara göre bazı üstünlükleri vardır. Bu özelliklerden ilki aynı yapının biri yüksek performanslı ve diğeri düşük maliyetli olan iki farklı FPGA üzerinde gerçeklenmesi ile kanıtlanan mimarinin ölçeklenebilirliğidir. İkinci olarak yapının esnekliği ve yeniden uyarlanabilmesi sıralanabilir. Bu sayede CNN şablonlarının boyu, tüm sinyallerin sabit noktalı aritmetikteki bit genişlikleri ve iterasyon sayısı gibi özellikler sentezleme öncesinde uyarlanabilmektedir. Üçüncü olarak şablon katsayıları, eşik değeri, sınır koşulları ve baypas modu gibi birçok parametrenin çalışma esnasında değiştirilebilmesini sağlayan programlanabilirlik özelliği verilebilir. Bu tez kapsamında önerilmiş olan CNN mimarisi literatürde tüm bu özellikleri bir araya getirdiği bildirilmiş olan tek CNN yapısıdır.

Anahtar Kelimeler: Hücresel sinir ağları, görüntü işleme, alanda programlanabilir kapı dizileri, gerçek zamanlı sistemler

CHAPTER 1

INTRODUCTION

1.1 Literature Review

Cellular Neural Networks (CNN) is a parallel computing paradigm [1] having many applications like image processing, artificial vision, solving partial differential equations, etc. A *p*–dimensional *q*–layer CNN structure consists of a *p*–dimensional spatial grid of neural cells and each cell contains *q* memory nodes and *q* inputs. The spatio–temporal dynamics of the system are tuned for specific tasks by defining local spatial synaptic interconnections between the neural cells. Generally, a 2–D 1–layer CNN structure with space invariant neural weights [2] is used in image processing applications, which is the focus of this thesis.

A Continuous–Time CNN (CT CNN) implementation [3, 4] has many advantages: it is fully parallel by its nature, its convergence rate is considerably faster then that of a digital implementation, it is easier to merge the architecture with an imaging sensor and obtain a focal plane processor to directly process the captured data as a pre–processor or artificial retina, etc. However, the highest implemented number of cells in a CT CNN processor is 176×144, to date, hence even a low–resolution input comparable to QVGA (320×240) may only be processed by tiling, i.e., dividing the image to smaller overlapped 'tiles' and process them individually [5]. Consequently, I/O bandwidth limit of a CT CNN processor makes it impossible to process a video stream like Full–HD 1080p@60 (1920×1080 resolution at 60 Hz frame rate) in real–time.

For a Discrete–Time CNN (DT CNN) implementation, first, a difference equation is obtained by discretization of the differential equation of a CT CNN. Then the difference equation may be solved on a software platform like a PC, DSP or GPU; or a custom hardware can be implemented as an ASIC or on an FPGA device. Software solutions are easier to design and modify while hardware implementations provide several orders of magnitude higher performance.

Using an FPGA device for a DT CNN implementation is preferable in most cases: it has very flexible parallel structures, its processing speed is second only to an ASIC implementation and it is cheaper than an ASIC solution. Consequently, the most notable DT CNN implementations [6, 7, 8] are implemented on FPGA devices, while [9] is implemented as ASIC. An alternative FPGA architecture of DT CNN was proposed in [10], which is named as Real–Time CNN Processor (RTCNNP, RTCNNP–v1). The architecture proposed in this thesis is a second–generation RTCNNP design called RTCNNP–v2 [11], [12]. Note that, in order to avoid confusion, the generic names of the proposed architectures, RTCNNP–v1 and RTCNNP–v2, are later renamed as *Steadfast–1* and *Steadfast–2*, respectively.

This is also worth stressing out that, this research was supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK), under project number 108E023, and a total number of four PhD theses are introduced from the project. The first thesis [13] is the foundation of the others, including this one, in which the *Steadfast–1* architecture was proposed. In the second thesis, a CNN based Gabor–type filter implementation is reported [14, 15]. Third, in this thesis, the *Steadfast–2* architecture is proposed, which also is the backbone of the second and fourth theses. Also note that, many common blocks of *Steadfast–2* and the Gabor–type CNN implementation proposed in [14] are designed as a team by the author of these theses. Finally, using the architecture proposed in this thesis to realize 2– or multi–layer CNN structures is the topic of the fourth thesis [16], which is still an ongoing work and expected to be finished soon.

Also note that, FPGA implementations of DT CNN are not limited to the ones referred in this thesis, however, the other structures reported in the literature are not designed to be general–purpose single–layer 2-D CNN emulators. For example, the architecture proposed in [17] is a class of DT CNN implementation, which is tailored for a specific task of active wave computing. These class of application–specific FPGA implementations are beyond the scoop of this thesis.

1.2 Aim of Thesis

A 2–D CNN structure is considerably suitable for image processing applications, as many image processing algorithms can be implemented on the same structure, eliminating the need to use mixed structures and continuously changing them for the needs of new applications. However, as mentioned in Section 1.1, the main bottleneck of CT CNN implementations reported in the literature is that, tiling should be used in order to process even the most basic resolutions like QVGA (320×240), hence they are not suitable for high–resolution real–time processing. On the other hand, even if some DT CNN implementations partly overcome this problem and has the ability to be used for resolutions up to VGA@60 (640×480 resolution, 60 Hz frame rate), they are still insufficient for modern resolutions like Full–HD 1080p@60, let alone for the military or aerospace applications where resolutions of the images are even higher.

Aim of this work is to design a real–time DT CNN implementation supporting not only higher frame–rates, but also higher resolutions, including Full–HD 1080p@60. Consequently, it will be possible to use CNN in image processing applications of most modern systems.

1.3 Original Contribution

As mentioned in Section 1.1, the *Steadfast–1* [13] structure (RTCNNP–v1) is the basis of the architecture proposed in this thesis. However, *Steadfast–1* is a static design, fixed to VGA@60 resolution and frame rate, with only pre–synthesis configurable template coefficients and bias. Furthermore, adding or changing any part of the design leads to a redesign process of the central processing unit, which makes the design inflexible, not reconfigurable and not reusable, ultimately making the design impractical.

The most original contribution of this thesis is the introduction of a local control structure for the pipelined CNN emulator arrays, hence making the new design almost infinitely flexible, reconfigurable and reusable. The local control structure makes it possible to design a pre–synthesis configurable architecture and easily describe it in VHDL. The second originality is the runtime programmability of the new architecture. The template coefficients, bias value and many other parameters are designed to be programmable, which makes the design practical to be used in image processing applications.

Finally, two prototypes are introduced on both a high–end and a low–cost FPGA device, capable of processing Full–HD 1080p@60 images at real–time, which makes the system the fastest CNN implementation, to date. Furthermore, processing speed of the high–end prototype is limited by the DVI I/O interface hardware, and the FPGA implementation is in fact faster by a factor of 2.5–3.

CHAPTER 2

THE CELLULAR NEURAL NETWORK STRUCTURE

In the most general case, a CNN structure is a p–dimensional q–layer spatial grid of neural cells, with each cell containing q memory nodes, each memory node having an input, and has space–variant local interconnections between cells. However, mostly m–neighborhood one–layer space–invariant continuous–space CNN structures are used in image processing applications, which is the focus of this thesis. A representation of a 2–D CNN grid and its local interconnections are given in Figure 2.1, where it is assumed that only the immediate neighbors are connected with each other, which is called a *one–neighborhood CNN*.

2.1 Mathematical Model of a Continuous–Time One–Layer Space–Invariant CNN

The Chua–Yang CNN model of an m–neighborhood one–layer space–invariant continuous–time CNN with $I \times J$ rectangular array of $C(i,j)$ cells is completely described in [2] by the cell state and output equation pair

$$\dot{x}_{ij}(t) = -x_{ij}(t) + \sum_{k,l=-m}^{m} \left(a_{kl} y_{i+k\,j+l}(t) + b_{kl} u_{i+k\,j+l} \right) + z, \tag{2.1}$$

$$y_{ij}(t) = f(x_{ij}(t)) = 0.5 \left(\left| x_{ij}(t) + 1 \right| - \left| x_{ij}(t) - 1 \right| \right), \tag{2.2}$$

where (i,j), $i \in \{1,2,...I\}$, $j \in \{1,2,...J\}$ are the spatial Cartesian coordinates, $x_{ij}(t)$ is the cell state at time t, u_{ij} is the constant–valued cell input, a_{kl} and b_{kl}, $k,l \in \{-m,...0,...m\}$, $m \in \mathbb{N}$ are the constant–valued feedback and input coefficients, respectively, z is the

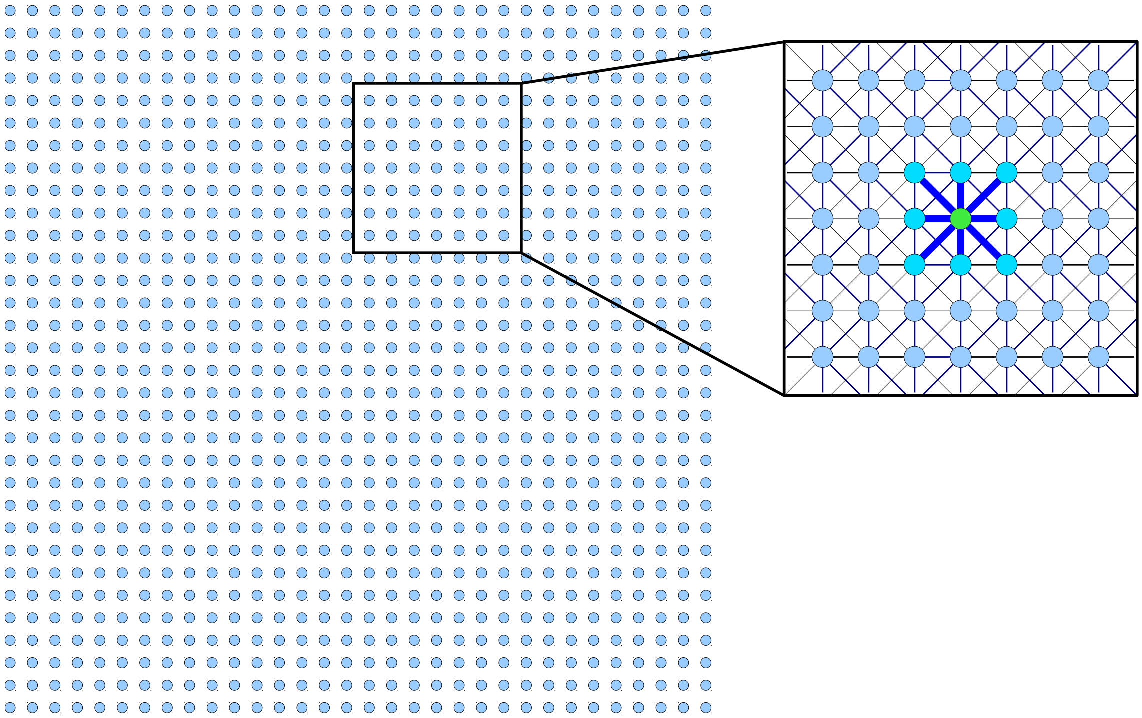


Figure 2.1 A 32 × 32 spatial grid of a CNN, a 7 × 7 section of the grid and its spatial interconnections

threshold value and y_{ij} is the cell output (Fig. 2.2a). Eq. (2.1) can be written as

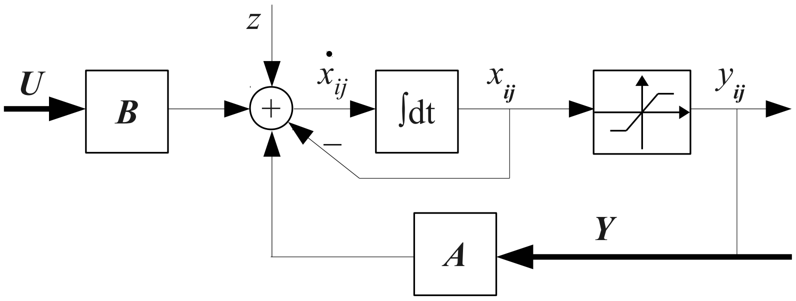
$$\dot{x}_{ij}(t) = -x_{ij}(t) + \boldsymbol{A} \circledast \boldsymbol{Y}_{ij}(t) + \boldsymbol{B} \circledast \boldsymbol{U}_{ij} + z, \tag{2.3}$$

where $\circledast$ is a convolution–like operator called *template–dot–product*, $\boldsymbol{A}$ and $\boldsymbol{B}$ are the feedback and feed–forward templates, $\boldsymbol{Y}_{ij}(t)$ and $\boldsymbol{U}_{ij}$ are the translated masked output and input, respectively. For $m = 1$

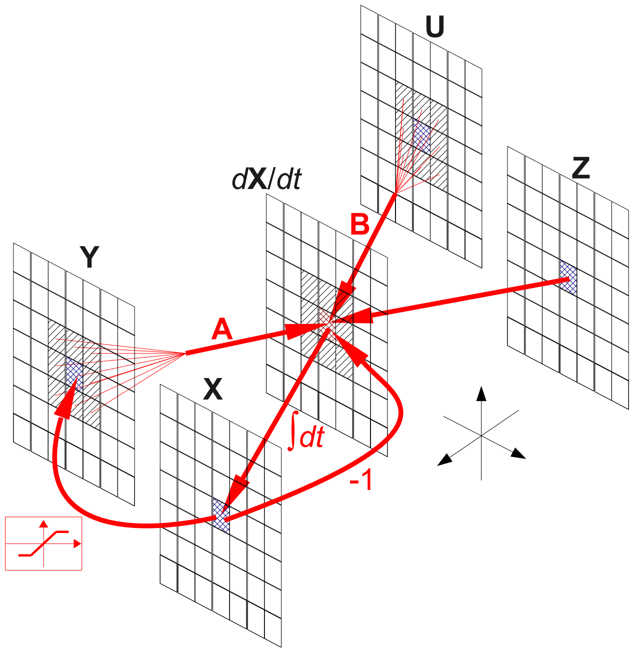
$$\boldsymbol{A} = \begin{bmatrix} a_{-1\,-1} & a_{-1\,0} & a_{-1\,1} \\ a_{0\,-1} & a_{00} & a_{01} \\ a_{1\,-1} & a_{10} & a_{11} \end{bmatrix}, \qquad \boldsymbol{B} = \begin{bmatrix} b_{-1\,-1} & b_{-1\,0} & b_{-1\,1} \\ b_{0\,-1} & b_{00} & b_{01} \\ b_{1\,-1} & b_{10} & b_{11} \end{bmatrix},$$

$$\boldsymbol{X}_{ij}(t) = \begin{bmatrix} x_{i-1\,j-1}(t) & x_{i-1\,j}(t) & x_{i-1\,j+1}(t) \\ x_{i\,j-1}(t) & x_{ij}(t) & x_{i\,j+1}(t) \\ x_{i+1\,j-1}(t) & x_{i+1\,j}(t) & x_{i+1\,j+1}(t) \end{bmatrix}, \qquad \boldsymbol{U}_{ij} = \begin{bmatrix} u_{i-1\,j-1} & u_{i-1\,j} & u_{i-1\,j+1} \\ u_{i\,j-1} & u_{ij} & u_{i\,j+1} \\ u_{i+1\,j-1} & u_{i+1\,j} & u_{i+1\,j+1} \end{bmatrix}.$$

A 3–D block diagram of a one–neighborhood CT CNN with 3 × 3 templates is given in Fig. 2.2b.



(a) 2–D block diagram of a CNN



(b) 3–D block diagram of a one–neighborhood CNN

Figure 2.2 Block diagrams of a CNN structure

2.2 Mathematical Model of a Discrete–Time One–Layer Space–Invariant CNN

The mathematical model of a Discrete–Time CNN (DT CNN) is obtained by sampling (2.3) and (2.2) in the time domain by

$$x_{ij}(t)\big]_{t=nT_s} = x_{ij}(nT_s) \triangleq x_{ij}(n)$$
$$\dot{x}_{ij}(t)\big]_{t=nT_s} = \dot{x}_{ij}(nT_s) \triangleq \dot{x}_{ij}(n)$$
$$y_{ij}(t)\big]_{t=nT_s} = y_{ij}(nT_s) \triangleq y_{ij}(n)$$

and applying Forward–Euler approximation

$$\dot{x}_{ij}(n) \cong \frac{x_{ij}(n+1) - x_{ij}(n)}{T_s} \tag{2.4}$$

to the time–derivative in (2.3), which yields the cell state and output equation pair.

$$x_{ij}(n+1) = x_{ij}(n) + T_s\big(-x_{ij}(n) + \boldsymbol{A} \circledast \boldsymbol{Y}_{ij}(n) + \boldsymbol{B} \circledast \boldsymbol{U}_{ij} + z\big), \tag{2.5}$$

$$y_{ij}(n) = f(x_{ij}(n)) = 0.5\left(\left|x_{ij}(n)+1\right| - \left|x_{ij}(n)-1\right|\right). \tag{2.6}$$

2.3 Mathematical Model of the Full Signal Range Model of a DT CNN

Although it is possible to implement (2.5) directly, Full Signal Range (FSR) model of DT CNN is easier to implement. The FSR model is originally proposed for analog CNN implementations, as in [18], where it is stated that any voltage in a chip does not exceed the rail voltages, hence the implemented CNN differs from the original Chua–Yang CNN model. In other words, physical voltage of a state node does not exceed $\pm 1V$, remaining in the *full signal range*. Consequently, all CT CNN implementations actually use the FSR model of CNN, and all CNN templates defined in the literature are designed work on both models.

Designers of most DT CNN implementations are inspired by the idea and applied the FSR model to a DT CNN, however, the method of obtaining the FSR model of a DT CNN is not clearly described in the literature. The new model is obtained by changing the difference equation given in (2.5) by defining

$$y_{ij}(n) \triangleq x_{ij}(n) \tag{2.7}$$

and modifying (2.6) to

$$y_{ij}(n+1) \triangleq f(x_{ij}(n+1)). \tag{2.8}$$

Note that, the operation is actually not about arranging a mathematical equation, but defining a new discrete–time model over the old one by modifying one section of a difference equation pair while keeping the other part as it is. Combining (2.7), (2.8) and (2.5), cell state equation of the FSR model of a DT CNN is obtained as

$$x_{ij}(n+1) = (1-T_s)y_{ij}(n) + T_s\boldsymbol{A} \circledast \boldsymbol{Y}_{ij}(n) + T_s\boldsymbol{B} \circledast \boldsymbol{U}_{ij} + T_s z,$$

which can be written as

$$x_{ij}(n+1) = \bar{\boldsymbol{A}} \circledast \boldsymbol{Y}_{ij}(n) + \bar{\boldsymbol{B}} \circledast \boldsymbol{U}_{ij} + \bar{z}, \tag{2.9}$$

where new template coefficients and threshold are defined by

$$\bar{a}_{kl} = \begin{cases} (1-T_s) + T_s a_{kl} & k,l = 0, \\ T_s a_{kl} & \text{otherwise,} \end{cases}$$

$$\bar{b}_{kl} = T_s b_{kl}$$

$$\bar{z} = T_s z.$$

Combining (2.9) and (2.8), output equation of the FSR model of DT CNN is obtained as

$$y_{ij}(n+1) = f\big(\bar{\boldsymbol{A}} \circledast \boldsymbol{Y}_{ij}(n) + \bar{\boldsymbol{B}} \circledast \boldsymbol{U}_{ij} + \bar{z}\big). \tag{2.10}$$

In a digital implementation, it is seen from (2.10) that it is no longer necessary to store $x_{ij}(n)$ as opposed to (2.5), as all information regarding $x_{ij}(n)$ is transferred to $y_{ij}(n)$. On the other hand, $y_{ij}(n)$ can be represented with less bits in fixed–point arithmetic, as $|y_{ij}(n)| \leq 1$, hence integer part of $y_{ij}(n)$ consist of only a sign bit, which means less memory. In other words, the idea is to let $x_{ij}(n+1)$ to grow during the computation process, then pass the final value from a saturator to obtain $y_{ij}(n+1)$, and finally store only $y_{ij}(n+1)$ for the next iteration while wiping $x_{ij}(n+1)$.

Note that, the expression ‘FSR model of DT CNN’ henceforth shortly referred to as ‘DT CNN’, as other mathematical models of DT CNN are beyond the scope of this thesis.

CHAPTER 3

CELLULAR NEURAL NETWORK IMPLEMENTATIONS

Implementing or using a Continuous–Time CNN (CT CNN) architecture over a traditional image processing structure has many advantages:

- CNN is a highly regular structure which makes it easier to implement;
- the spatio–temporal dynamics of CNN is well defined with a mathematical model, as opposed to many image processing algorithms based on empirical results;
- several image processing tasks can be realized on the same CNN structure by simply changing the templates, bias, initial conditions and boundary conditions;
- and the computation is carried out very fast due to the parallel structure of CNN.

However, a considerable implementation difficulty is introduced as the input image gets larger, and implementing a larger grid is either impossible or not feasible after a certain point. Consequently, grid size of the largest CT CNN implementation is 176×144, to date. On the other hand, none of the general–purpose Discrete–Time CNN (DT CNN) implementations are reported to be capable of working on images larger than 640×480 resolution with 60 Hz frame rate, in real time, except for the previous publications of this thesis [11, 12], which proves the added value of this thesis.

In this chapter, the most notable general–purpose CT CNN and DT CNN implementations of the literature are summarized and their implementation methods are discussed.

3.1 Continuous–Time CNN Implementations

Continuous–time implementation of a 2–D CNN is relatively straightforward: the 2–D grid of a CNN is directly transferred to an analog chip. A 32×32 CNN grid is given in Figure 2.1, where each cell contains a capacitive (analog) memory node and spatial interconnections. Circuit model of a $C(i, j)$ cell and simplified block diagram of a CT CNN implementation are given in Figure 3.1 and 3.2, respectively.

3.1.1 CT CNN Implementation Examples

The most notable CT CNN implementations are ACE16K [3] and Eye–RIS [4], whose grid sizes are 128×128 and 176×144, respectively. Both implementations are CNN Universal Machines (CNN–UM), that is, they are designed to be stored programmable array computers for implementing sequences of template operations with local analog and logic memory [2]. In other words, they are implemented not only to compute a single CNN equation, but also store/reload their outputs as intermediate results to realize complex tasks. For example, an enhanced edge detection algorithm can be implemented by: saving an input, applying dilation operation to the input and saving the result, applying erosion operation to the input and saving the result, carry out an XOR operation between two results and relay the final result to the output.

3.1.2 Processing Large Images with Smaller Grids

Implementing a CT CNN grid larger than 176×144 is not feasible, hence larger images are processed with a method called *tiling*, i.e., dividing the image to smaller pieces called *tiles*, whose sizes are the same or smaller than that of the grid, and processing them individually. Some possible tiling schemes are given in Figure 3.3. The tiles should be overlapped to eliminate boundary effects: overlapping one–pixel may be sufficient for a class of DT CNN implementation, but at least a few pixels should be overlapped for CT CNN. Note that, the amount of overlapping depends on the CNN templates that will be realized on an implementation, hence it may be necessary to be excess if the template relates pixels farther apart.

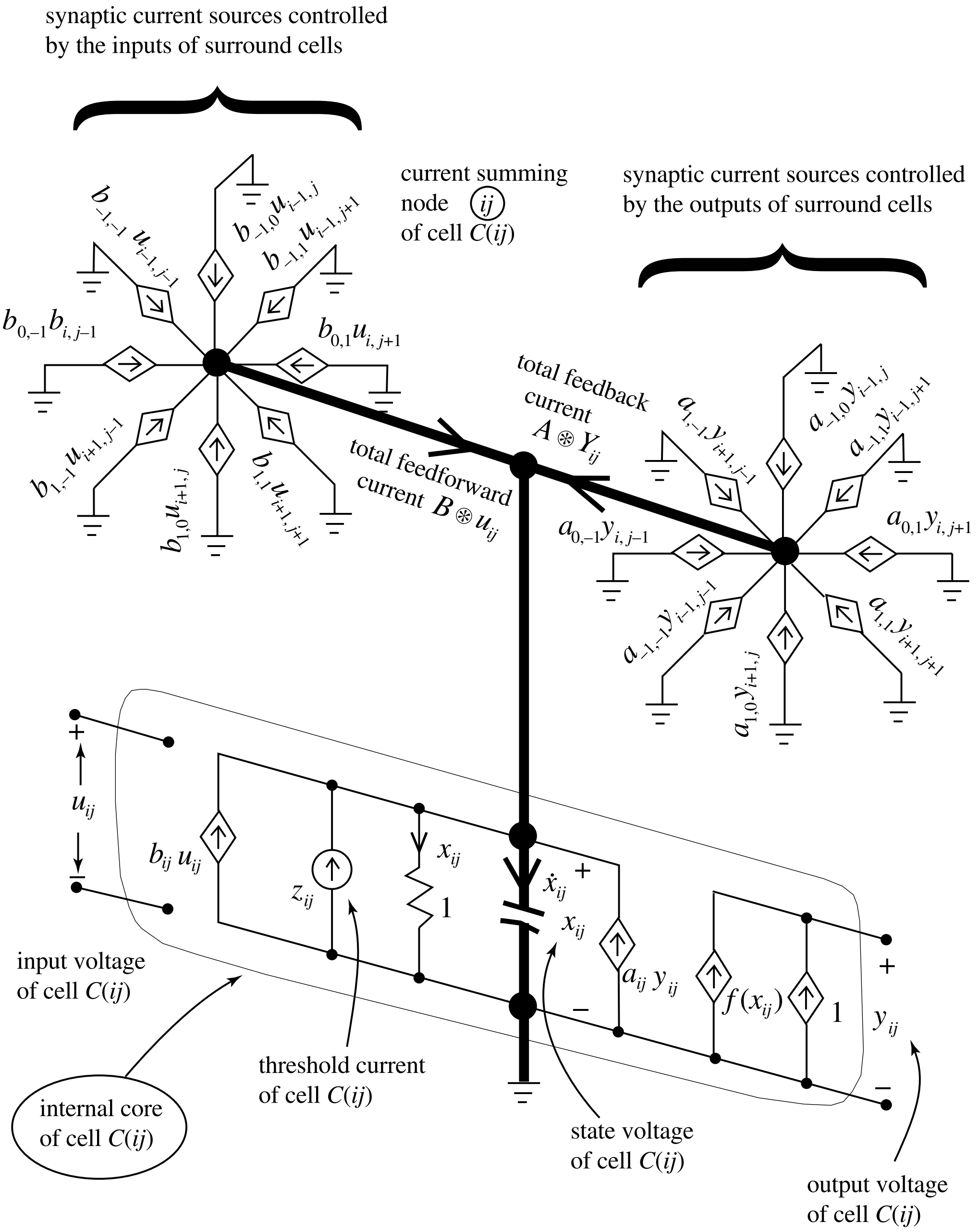


Figure 3.1 Analog circuit model of a CT CNN cell [2]

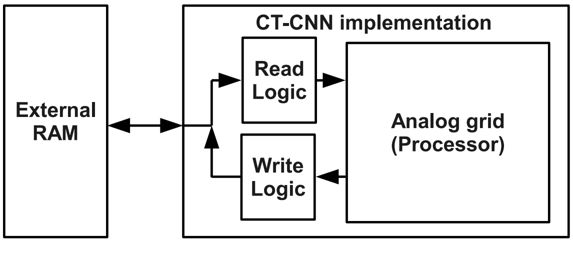
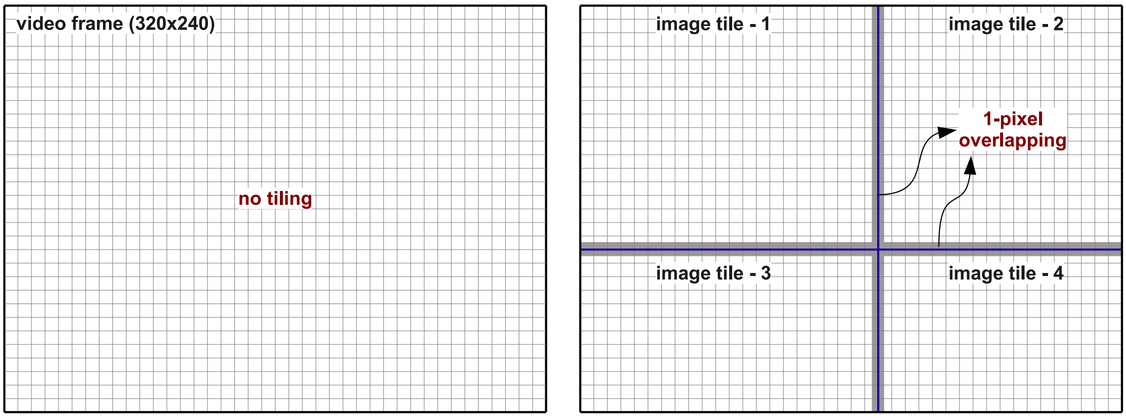
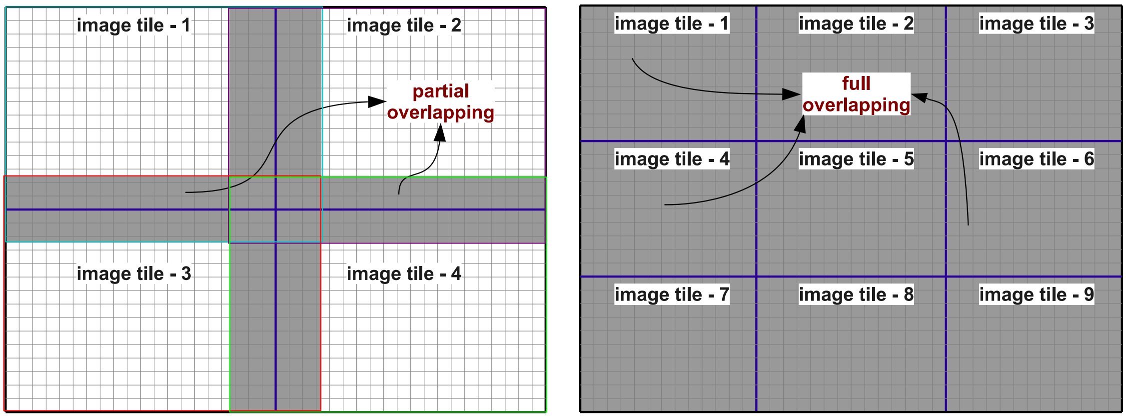


Figure 3.2 Block diagram of a CT CNN implementation



(a) No tilling (b) One–pixel overlapping, only suitable for a pipelined DT CNN implementation



(c) Partial overlapping, suitable for most CNN implementations (d) Excess overlapping, rarely required

Figure 3.3 Tiling schemes

For example, two CNN simulations are carried out on a PC with grid sizes of 176×144 and 320×240, where templates of a Gauss–type CNN low–pass filter are chosen, and a 320×240 image is processed with and without tiling (Figure 3.4). The original image and the expected result of the Gauss–type filter are given in Figure 3.4a and 3.4b, while the results for insufficient and sufficient overlapping are obtained as in Figure 3.4c and 3.4d, respectively.

However, while partial or excessive overlapping schemes are suitable for many CNN templates, some may be impossible to realize by tiling. For example, *global connectivity detection* [19] templates are designed to delete open and one–pixel wide curves as seen in Figure 3.5a and 3.5b, yet even a properly overlapped tiling scheme with 22×22 tiles gives a different result (Figure 3.5c).

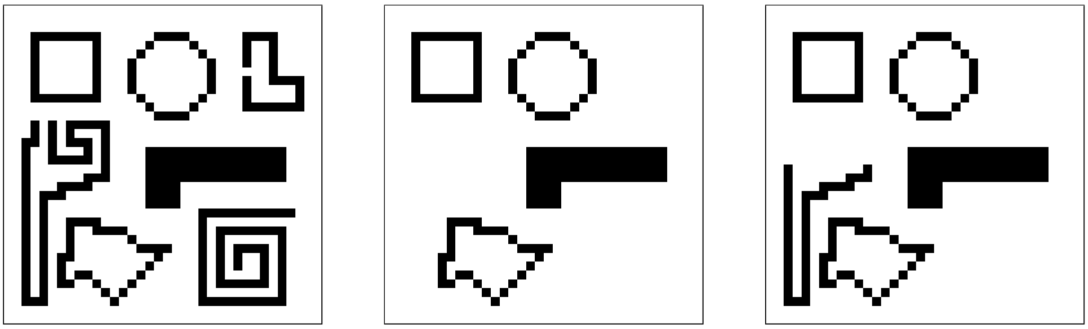
(a) A 320 × 240 test image

(b) Full–frame output of a CNN Gauss filter

(c) Tiled output with one–pixel overlapping

(d) Tiled output with partial/excess overlapping

Figure 3.4 A tiling example that shows the input and results of a CNN Gauss filter simulated for three different tiling schemes: full–frame, one–pixel overlapped and partially overlapped



(a) A 36 × 36 test image

(b) Intended output

(c) Tiled output with a proper overlapping

Figure 3.5 Another tiling example for global connectivity detection templates, where tiling is failed

In short, a CT CNN implementation has some shortcomings. First, grid size is limited by some feasibility issues of the analog IC technology. Moreover, tiling is not always reliable for some CNN templates, hence these networks can only be simulated or emulated on a digital platform for large images. Second, bit depth of a CT CNN is limited to 7 bits due to the electrical noise and crosstalk of an analog implementation. Consequently, even obtaining a regular 256 level gray–scale result is not possible with CT CNN. Finally, as opposed to a digital implementation, modifying an analog IC design is a very comprehensive work, which can almost be considered as a new project. As a result, digital implementations of CNN are preferable in most cases.

3.2 Discrete–Time CNN Implementations

A CT CNN implementation is a fully–parallel analog processor array by its nature. On the other hand, the difference equation (2.10) can only be solved by multiple iterations. Consequently, fully–parallel implementation method described in Section 3.1 is not applicable to a DT CNN. Note that, it is still possible to implement a fully parallel iterator with dedicated memory and computation resources assigned to each cell, however a tremendous amount of computation resources are required for such a design.

The most basic digital implementation of a CNN is a simulation on a processor–based platform like a PC. Considering that template–dot–product operator is actually a convolution–like operator, calculating one iteration of (2.10) means computing two convolutions and summing the results and the bias. The computation can be carried out by raster scanning the input and output images (matrices) $\mathbf{U}$ and $\mathbf{Y}$, respectively, i.e., scanning the matrices in the order given in Figure 3.6, and computing outputs of each cell one by one. The result of an iteration is computed at the end of the raster scan and the operation is repeated N times, which is the number of Euler iterations desired. The processing work–flow can be summarized like the following:

1. set line and column indexes to the first cell,
2. read inputs and outputs from the cells being in m–neighborhood of the given cell,

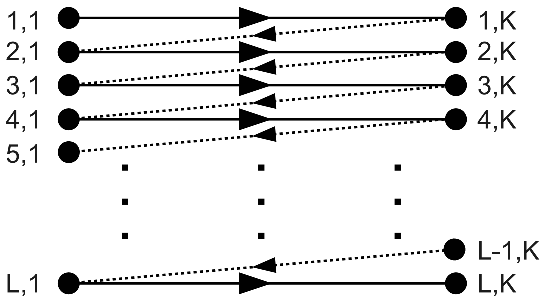


Figure 3.6 Row–wise packing scheme of raster scanning

3. perform the template–dot–product and addition operations,
4. save the result,
5. if not the last cell, set indexes to the next cell; else, set indexes to the first cell for the next iteration,
6. go to step 2.

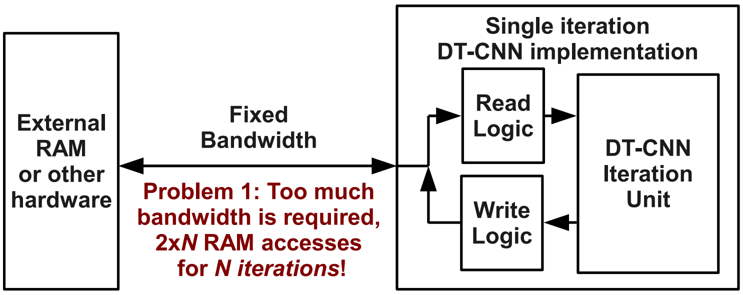
Note that, computing all iterations in a loop is extremely time consuming, and parallel or pipelined processors should be used for most real–time image processing tasks. Consequently, the computation process should be divided to sub–processes in order to make it suitable for multiple processors.

3.2.1 Hardware Implementation Methods of DT CNN

The processing work–flow can directly be implemented on a digital hardware like an FPGA (Figure 3.7a). Note that, even if a tiling scheme is used, then just the intermediate results are tiled instead of the final results, which only corresponds to change the computation order. Consequently, full-frame processing or tiling does not affect the final result in any way, which is not the case of an analog implementation (Figure 3.7b and 3.7c). However, new problems are introduced with a digital implementation of CNN:

Problem 1: Arises when too much I/O access is required from/to external hardware or RAM to read/write intermediate computation results.

Problem 2: Resources of the hardware may be insufficient for the implementation of multiple processors,



(a) A DT CNN implementation with a single iteration unit

(b) Full–frame processing result (c) Tiled output with one–pixel overlapping

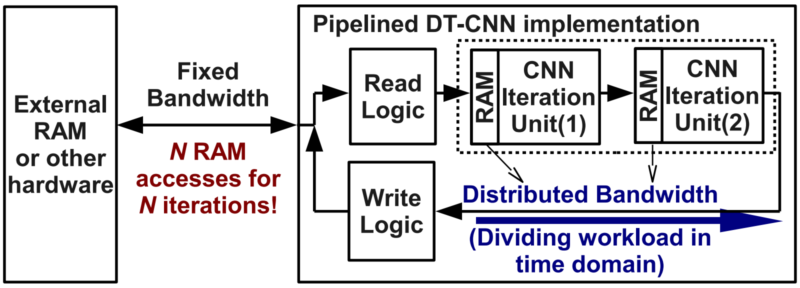
Figure 3.7 Block diagrams of a DT CNN implementation with a single iteration unit, and simulation results of the implementation

> **Problem 3:** Caused when the input pixel rate is higher than the maximum operating frequency of the hardware resources.

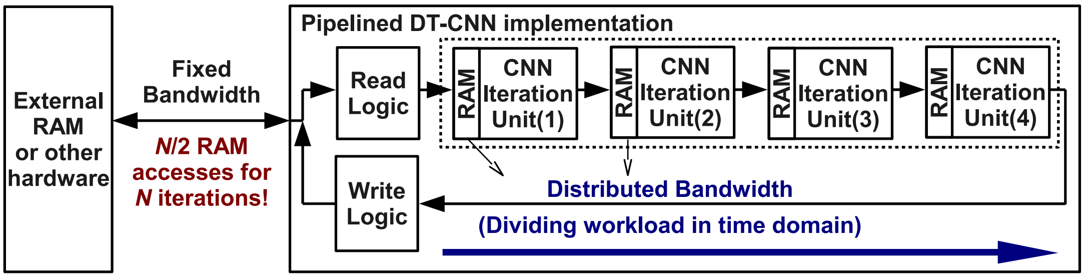
The first and third problems are solved by using multiple processors and dividing the computation in temporal and spatial domains, respectively, while processors are distributed among many hardware units to solve the second problem.

3.2.1.1 Dividing the Computation in the Temporal Domain

The first problem concerns memory bandwidth of the external RAM unit: performing N iterations means accessing the same memory locations N times to read and N times to write, $2N$ in total, as opposed to only 2 of an analog design. The solution is to use a pipelined processor array instead of a single iterator, which corresponds dividing the spatio–temporal computation flow in the temporal domain, hence the bandwidth requirement is divided by the number of iteration units. For example, if 2 and 4 processors are pipelined, the required bandwidth will be N and $N/2$, respectively, as opposed to $2N$ of



(a) A DT CNN implementation with two pipelined iteration units



(b) A DT CNN implementation with two pipelined iteration units

Figure 3.8 Pipelining in a DT CNN implementation: dividing the workload in time domain

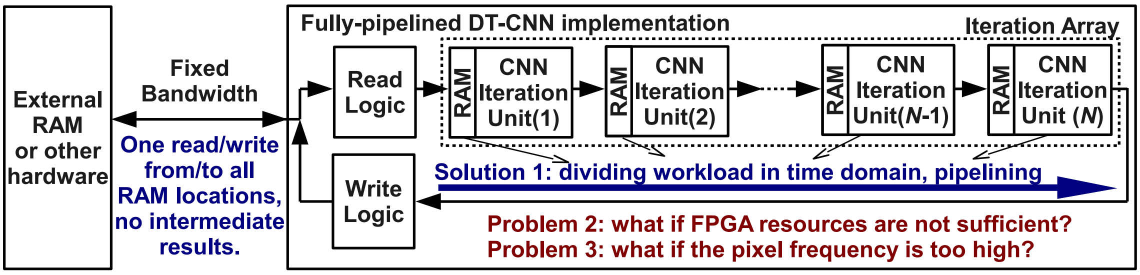


Figure 3.9 A fully–pipelined DT CNN implementation

the single processor scheme (Figure 3.8).

The ultimate solution to the first problem is to make the design fully–pipelined, i.e., adding as much iteration units as the processing requires, which is called unrolling the iterations. In other words, a processor array containing N processors can be implemented on hardware to completely eliminate excess memory accesses as given in Figure 3.9, where the output of the last iteration unit is the final result. Fully–pipelining solves the memory bandwidth problem while introducing the second problem: what if the hardware resources are not sufficient to implement N processors?

The second problem is solved by using multiple digital hardware units, e.g., using multiple FPGA devices to implement a longer pipeline (Figure 3.10). Note that, the bandwidth

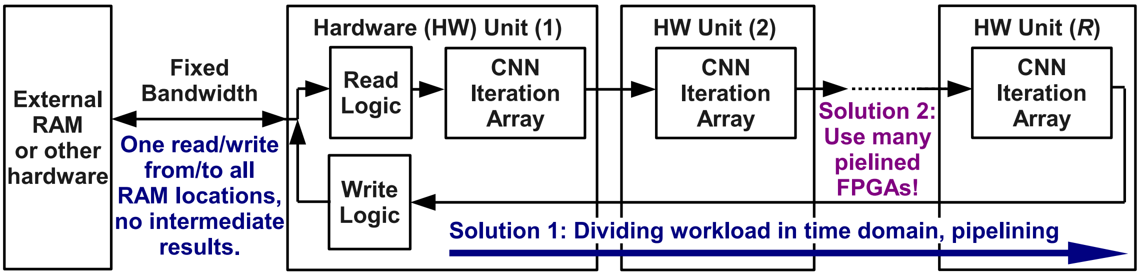


Figure 3.10 A fully–pipelined DT CNN implementation with multiple hardware units

of the intermediate data flow between hardware units may be slightly higher than that of the main input, because in fixed point arithmetic the intermediate result should generally be represented with higher number of bits then the input for accuracy. However, in most cases it is trivial to customize the intermediate bandwidth, hence it is not a serious problem.

3.2.1.2 Dividing the Computation in a Spatial Domain

The third and the final problem rises when the input data rate is faster than the upper frequency limit of the internal resources of the digital hardware. For example, the pixel rate of a 4K@60 (3840 × 2160 resolution at 60 Hz frame rate) video signal is approximately 594 MHz, which is above or too close to the maximum operating frequency of any state of the art FPGA device, including the high–end products. Moreover, this problem can not be solved by pipelining, as we can show by analogy that the problem is not about the length of the pipeline, but the cross–section of it. In this case, adding a second pipeline parallel to the first one solves the problem, hence the solution is parallelism (Figure 3.11). There are several methods to make the computation parallel, however, considering that images are packed row–wise (Figure 3.6) in most cases, the best way is to divide the image to vertical stripes and process each stripe with a separate pipeline (Figure 3.12). Note that, with this method, the computation workload is divided along a spatial domain instead of the time domain.

The stripes should overlap with each other one pixel on both edges in order to avoid boundary effects. However, it is not sufficient to overlap only the input stripes, but intermediate results of all iterations should also be overlapped. Consequently, each iteration

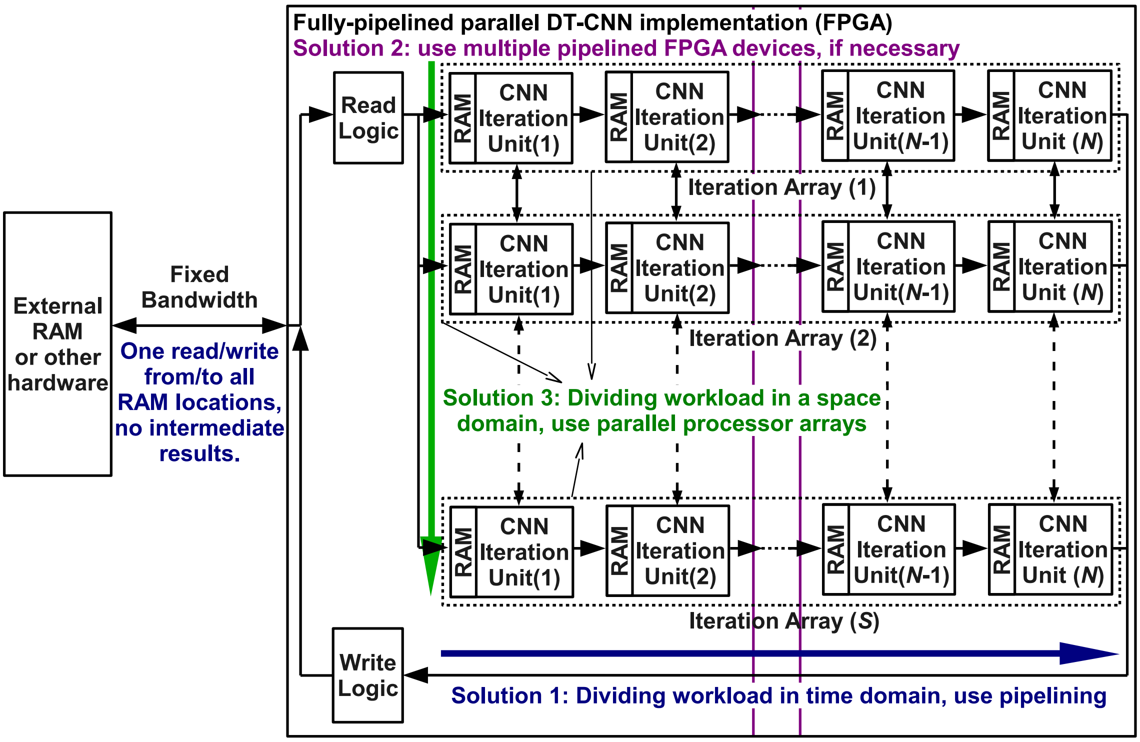


Figure 3.11 A fully–pipelined DT CNN implementation with parallel iteration arrays, where solutions of all three basic digital implementation problems are covered

unit should communicate with its spatial neighbor in order to send and receive the boundary values.

It is also worth noting that, there are many possible configurations of pipelining and parallelization while implementing the discussed methods of dividing the computation workload in the temporal and a spatial domain, respectively. A few practical examples are given below, where it is assumed that we have an FPGA device that is capable of holding up to 100 iteration units (processors), and each processor has an upper operating frequency of 300 MHz.

Example 1 *How to process a* 1080*p*@60 *video signal for* 250 *iterations?* The pixel frequency of a 1080p@60 video signal is 148.5 MHz, which is lower than 300 MHz, the maximum operating frequency of a processor, hence parallel processing is not required. However, at least $\lceil 250/100 \rceil = 3$ FPGA devices should be used to implement 250 iterations (Figure 3.13a).

Example 2 *How to process a* 4*K*@60 *video signal for* 40 *iterations?* The pixel frequency of a 4K@60 video signal is 594 MHz, higher than 300 MHz, consequently, at

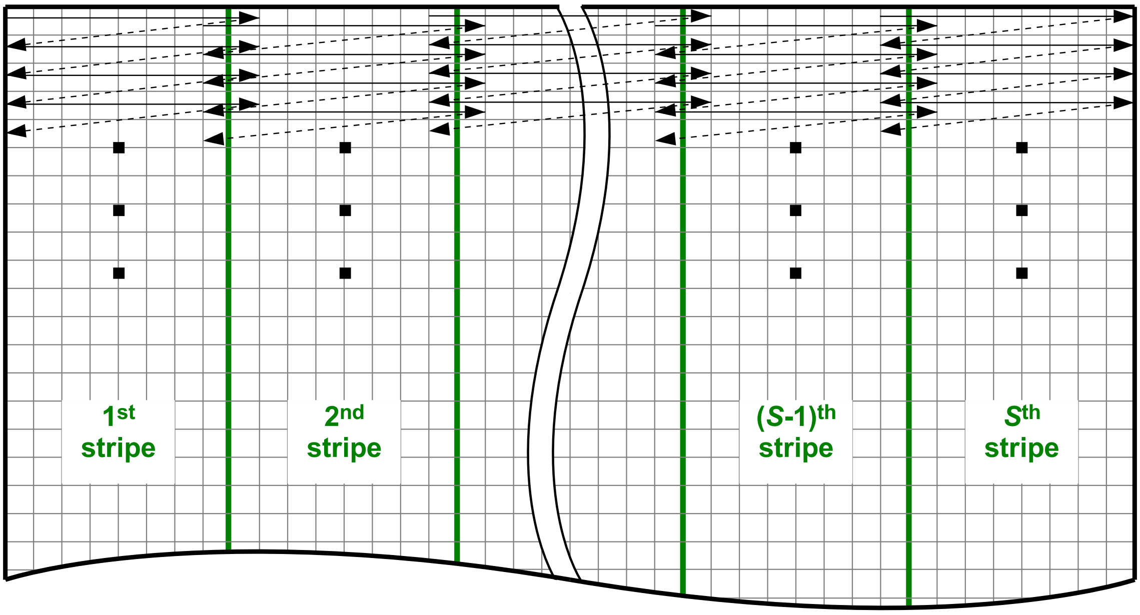


Figure 3.12 A parallelization scheme suitable for the processing of a row–wise packed image

least $\lceil 594/300 \rceil = 2$ stripes are required. In this case, $2 \times 40 = 80$ pipelined processors are necessary, which means using a single FPGA device is sufficient (Figure 3.13b).

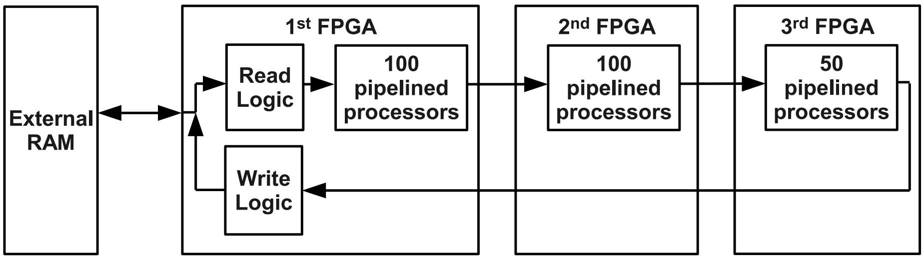
Example 3 *How to process a* 8K@60 *video signal for* 40 *iterations?* The pixel frequency of a 8K@60 video signal is 2.37 GHz, hence at least $\lceil 2370/300 \rceil = 8$ stripes are required. As each stripe requires 40 pipelined processors, at least $\lceil 8 \times 40/100 \rceil = 4$ FPGA devices should be used. Although there are many possible configurations, a possible solution is to divide the number of processors equally between four FPGA devices a given in Figure 3.13c.

3.2.2 DT CNN Implementation Examples

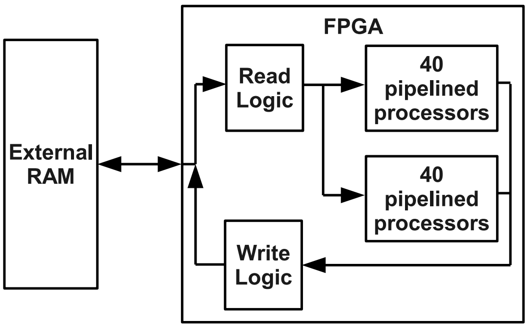
There are many DT CNN implementations of CNN, however, most of them are experimental and far from being usable in image processing tasks. Consequently, only the most notable DT CNN implementations are summarized in this section.

3.2.2.1 Implementation of Zarandy et al. (CASTLE)

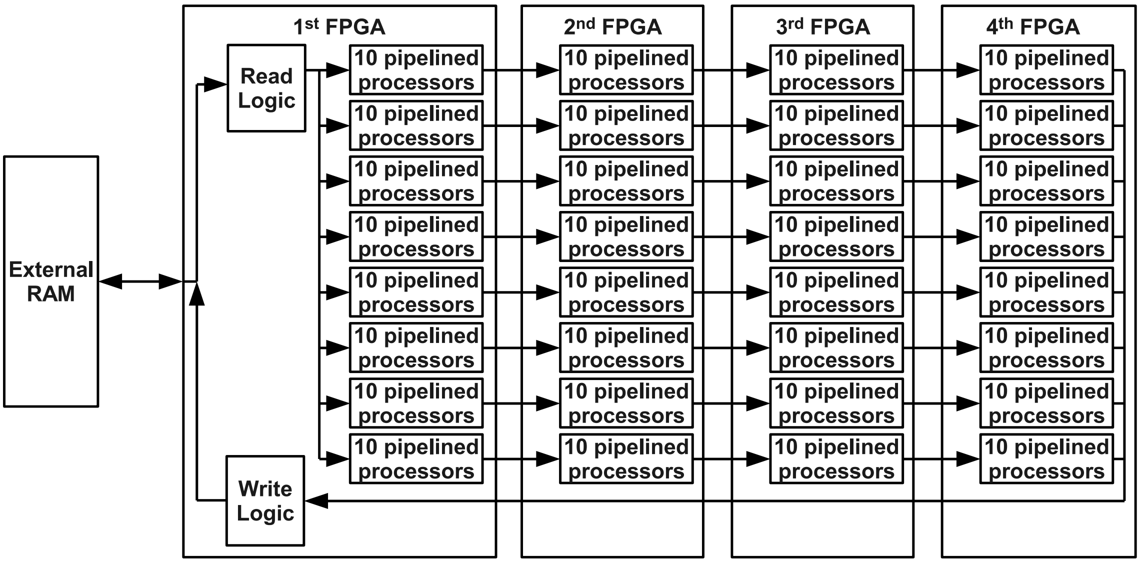
The first notable DT CNN implementation is CASTLE [9], an ASIC implementation, where both partial pipelining and parallelization schemes are used. In this design, a $K \times L$



(a) A solution of example 1



(b) A solution of example 2



(c) A solution of example 3

Figure 3.13 Possible hardware solutions of the examples

processor matrix can be implemented, where *K* is the number of iterations unrolled and *L* is the number of vertical stripes that the input image, consequently the cell array, is divided to (Figure 3.14). The pipelining scheme used in CASTLE is not full, i.e., iteration loop is not fully–unrolled, hence one intermediate iteration result out of *K* iterations are saved/loaded to/from an external memory unit.

Internal structure of a CASTLE processor is given in (Figure 3.15). The processor has three front input buses for the states $x_{ij}(n)$, constants g_{ij} and template select words Ts_{ij}. g_{ij} is the part of (2.10) which is constant through the Euler iterations:

$$g_{ij} = \bar{\boldsymbol{B}} \circledast \boldsymbol{U}_{ij} + \bar{z} \tag{3.1}$$

which is computed once for every pixel of each input image and carried as a constant through all Euler iterations. Consequently, it is sufficient for each processor to perform one template–dot–product operator for each iteration instead of two. Template select word is an indicator that is used to select one of the 16 templates stored in the template memory, which can be used to implement space–variant templates. The I/O busses LBUS and RBUS are used to communicate with the neighboring processors to give and take the boundary values.

A CASTLE processor stores a three line belt of the input state as shown in Figure 3.16, as states from one upper and one lower lines are required for the computation of a template–dot–product operation, for one neighborhood CNN ($m = 1$). Contents of each line buffer is copied to the next one at the end of each line.

Arithmetic unit of CASTLE is given in Figure 3.17, which is designed to perform a 3×3 template–dot–product operation in three clock cycles. Three states and template coefficients, S and T, respectively, are selected from the internal buffers of the processors at each clock cycle and multiplied by each other. Consequently, nine multiplications of a 3×3 template are carried out in three clock cycles. ACC/ACT registers are master/slave

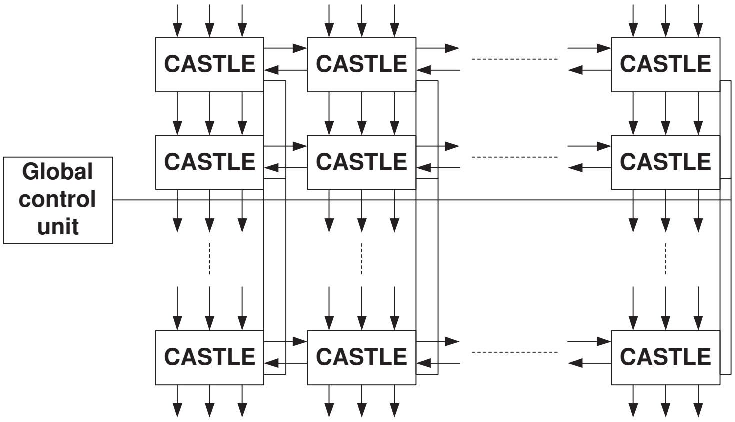


Figure 3.14 Processor organization of the CASTLE architecture [6]

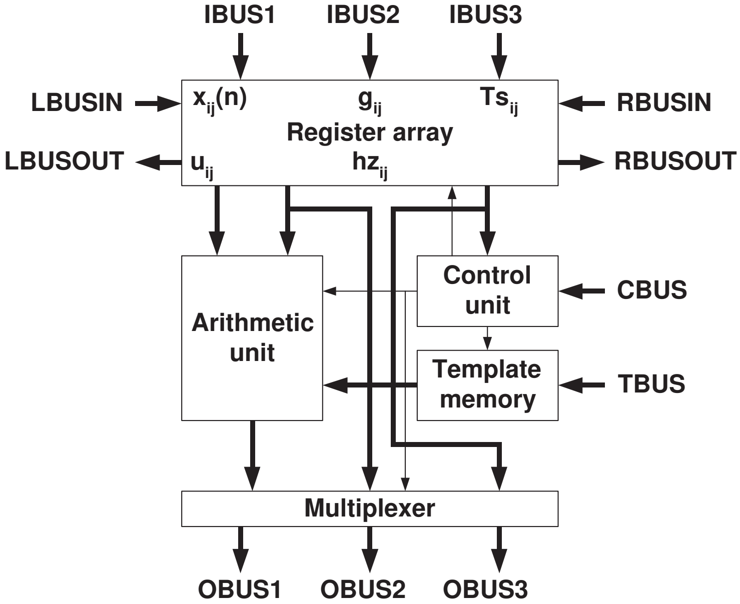


Figure 3.15 Block diagram of a CASTLE processor [6]

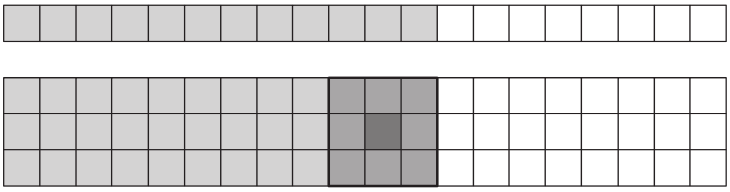


Figure 3.16 Memory belt stored in a CASTLE processor [6]

registers used to supply either the constant g, or intermediate result of the same addition operation computed at a previous clock cycle to the adder tree. Finally, the result is shifted, rounded and relayed to the output.

CASTLE has a considerably fixed architecture, as it is targeted for ASIC implementations. Only 3×3 templates are implemented with limited space–invariance support; although a CASTLE architecture with 5×5 templates is proposed in [20], but is not reported as implemented. Direct implementation of a multi–layer CNN is also not possible on CASTLE. Furthermore, precision of its arithmetic operations are programmable to 1, 6 or 12 bits of resolution, which is not sufficient for many CNN implementations [6].

3.2.2.2 Implementation of Nagy and Szolgay (Falcon)

Falcon [6] is an improved CASTLE architecture, implemented on an FPGA device. The processor organizations of Falcon and CASTLE are the same, hence again, a $K \times L$ proces-

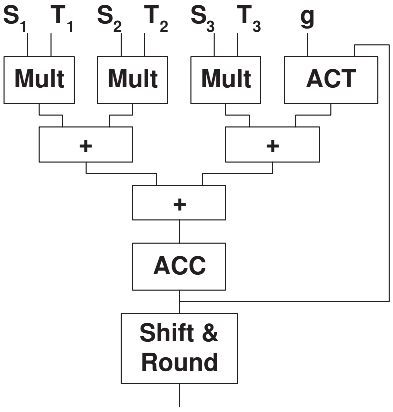


Figure 3.17 Arithmetic unit of a CASTLE processor [6]

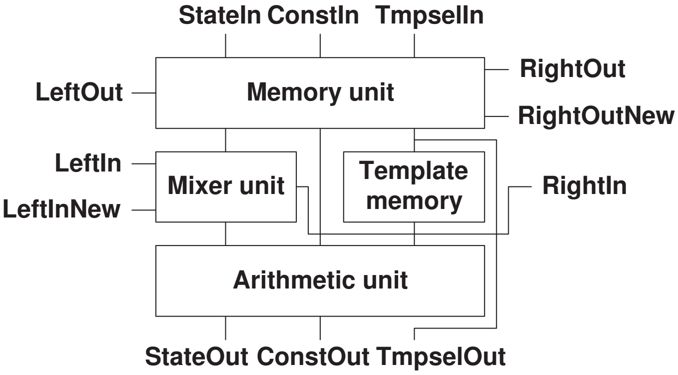


Figure 3.18 Block diagram of a Falcon processor [6]

sor matrix is implemented (Figure 3.14). Falcon has a very flexible computation structure, as opposed to CASTLE, which can be configured to realize multi–layer or space–variant CNN structures. As a result, it is considerably easy to configure Falcon to solve partial differential equations, or use it as a CNN–UM.

Block diagram of a Falcon processor unit is given in Figure 3.18, which is an improvement over the original CASTLE processor. First, a new left to right I/O bus is added to increase the control over the boundary conditions. Second, line buffers of the memory unit are replaced with shift registers as shown in Figure 3.19, which saves time by eliminating the process of copying contents of each line buffer to the next one. The multiplexed design makes it possible to implement zero–flux boundary conditions. Third, a more complex mixer unit is used to select the necessary states from line buffers and boundaries and relay them to an arithmetic unit (Figure 3.20). Finally, a pipelined Falcon arithmetic unit is designed as shown in Figure 3.21.

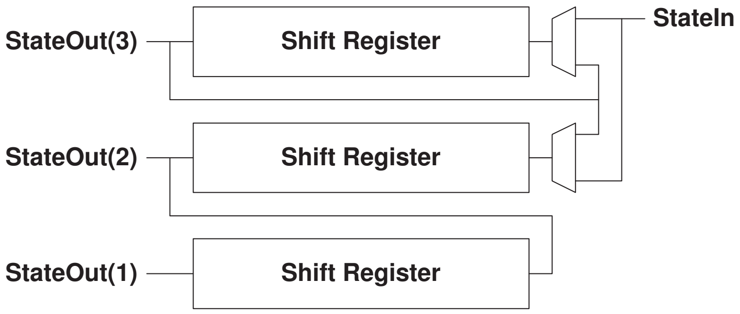


Figure 3.19 Block diagram of the memory unit of a Falcon processor [6]

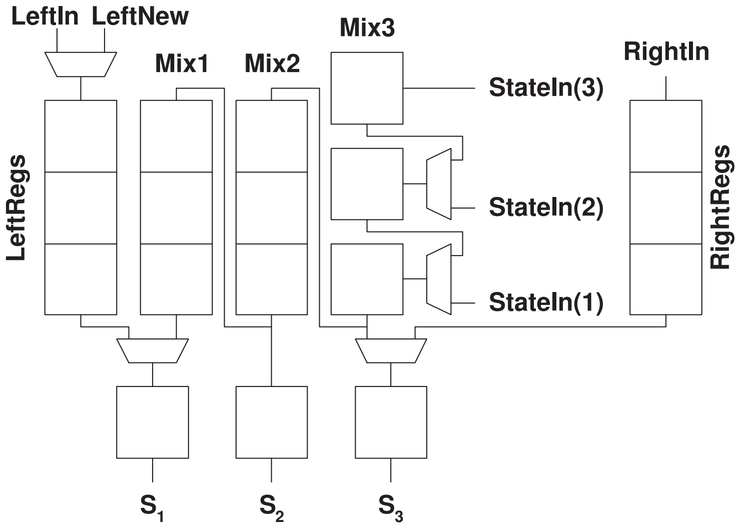


Figure 3.20 Block diagram of the mixer unit of a Falcon processor [6]

It is also proposed in [6] to modify the architecture of a Falcon processor to support $m = 2$ and $m = 3$ neighborhoods; 5×5 and 7×7 templates, respectively; however, it is not reported to be implemented as a working prototype as of to date.

The bird's–eye view of the implemented Falcon emulated CNN system is given in Figure 3.22, which is a CNN–UM implementation running by a host computer. The image I/O and control signals are merged in a host bus. A host interface control unit is responsible to control the main control unit and write/read the input and output to/from the external memory.

The most recent and the most capable Falcon implementations are reported in [21] and [22]. The architecture reported in [21] include a modified Falcon processor and the whole system is capable of processing full–HD 1080p@50 (1920×1080 resolution, 50 Hz frame rate) image streams in real–time and give a collision detection output. However, only the

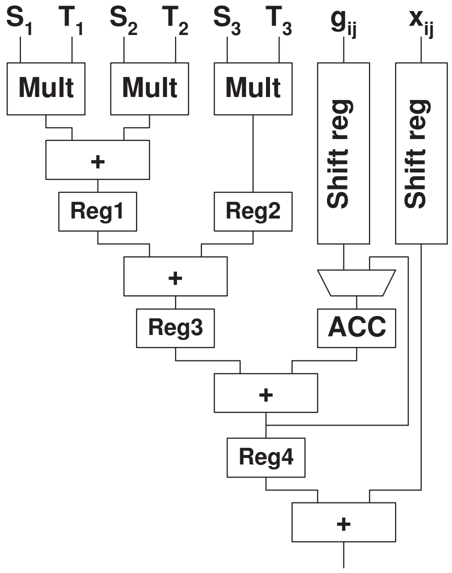


Figure 3.21 Block diagram of the arithmetic unit of a Falcon processor [6]

pre–processor part of the system performs at the given resolution and frame rate, and the Falcon core only processes a 128 × 128 part of the image. On the other hand, only frame rates are discussed in the second paper reported in [22], which lacks the information about the resolution. In short, none of the Falcon implementations are reported to operate at higher resolutions and frame rates than VGA@60 (640 × 480 resolution at 60 Hz frame rate).

3.2.2.3 Implementation of Malki and Spaanenburg

Malki and Spaanenburg proposed two main DT CNN implementations, where (3.1) is computed like in the case of CASTLE and Falcon, then multiple iterations are carried out. The first architecture reported in [23] has a similar approach with Falcon, with a different architecture. However, the reported pixel throughput of 180 Mpix/s is given as a simulation result, which is also a unrealistic estimation for a Virtex II 6000 FPGA kit; consequently, it is not clear that whether it is implemented as a working prototype, or not. Moreover, as it is stated in Section 3.2.2.2, MAC per second is a better criteria for performance comparison, which is not used in [23], either.

The second implementation is based on packet switching instead of pure pipelining [7]. In this method, the cell grid is divided to five types of cells; A, B, C, D and E; and the cells

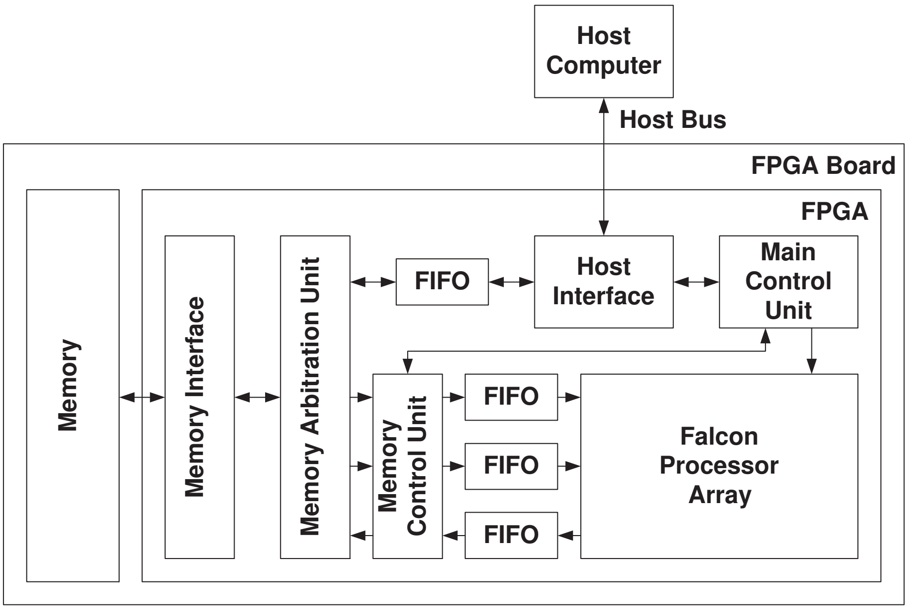
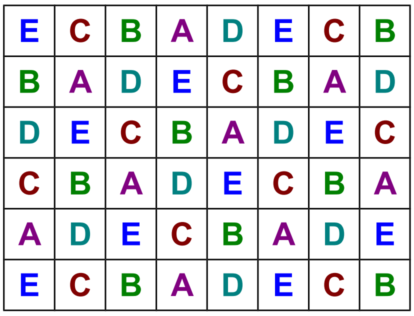


Figure 3.22 CNN UM implementation of a Falcon processor array [6]



E	C	B	A	D	E	C	B
B	A	D	E	C	B	A	D
D	E	C	B	A	D	E	C
C	B	A	D	E	C	B	A
A	D	E	C	B	A	D	E
E	C	B	A	D	E	C	B

Figure 3.23 The knight–placement of the neighboring cells [23]

that have a *knight jump distance* with one another is labeled as the same type (Figure 3.23). On the other hand, each cell is designed to have two operating modes: computation and communication. At any given time, only one type of cell group is activated for processing, while the others are set to communicate with the active cells to supply the data needed for computation. Communication is carried out by message packets containing data and row/column addresses. This method is reported to give more flexibility over the first method proposed by Malki and Spaanenbug, however, throughput is 50 times lower than that of the first design.

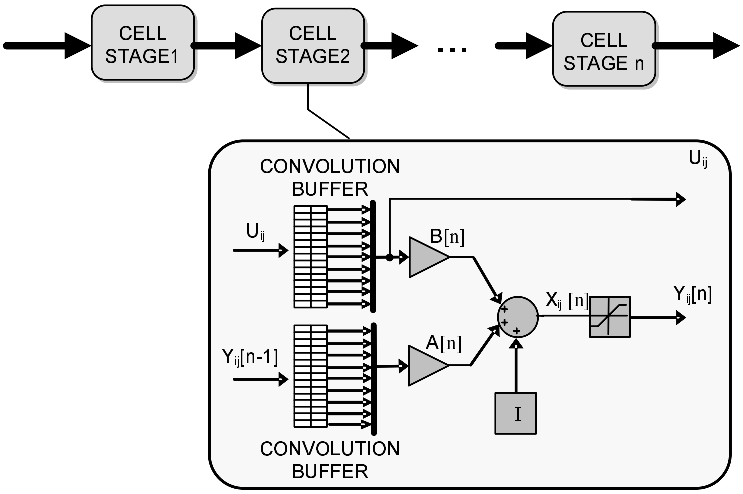


Figure 3.24 Processor array proposed by Martínez–Alvarez et al. [8]

3.2.2.4 Implementation of Martínez–Alvarez et al.

Martínez–Alvarez et al. have a DT CNN implementation reported in [8]. They have fully unrolled all Euler iterations, in other words, designed a fully pipelined architecture as described for Figure 3.9. Block diagram of their processor array and signal flow graph of a processor are given in Figure 3.24. The main difference of this design is that, each processor computes (3.1) instead of getting the result from a preprocessor, which doubles the number of multiplication and addition operations compared to the other implementations. In the original design, only one multiplier is implemented for the arithmetic part of the processor, responsible to calculate all multiplication results of two template–dot–product operations. Time sharing is used in the computation, which is carried out in 18 clock cycles, as there are 18 coefficients. The internal structure of the processors are also designed to be fully pipelined to avoid any wait cycles, which means that the processing clock frequency should be 18 times the pixel clock frequency.

Martínez–Alvarez et al. also proposed a class of multi–FPGA implementation of DT CNN, similar to the one given in Figure 3.10 [24]. It is also worth noting that the emulator is the first DT CNN implementation that is reported to be capable of processing VGA@60 video images in real–time. However, the fixed architecture of the design makes it difficult to modify the architecture to support higher resolutions at the same frame rate. Consequently, the frame rate should be lowered to increase the resolution, keeping the

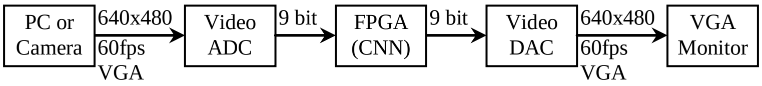


Figure 3.25 Block diagram of a *Steadfast–1* prototype [13]

pixel rate below the maximum operating frequency of processors, hence the the highest resolution and frame rate are reported as $1024 \times 1024@22$, to date.

3.2.2.5 Implementation of Kayaer and Tavsanoglu (Steadfast–1)

Kayaer and Tavsanoglu have proposed a modified DT CNN structure in [10, 13], which was first called as Real–Time Cellular Neural Network Processor (RTCNNP, RTCNNP–v1), which is later renamed as *Steadfast–1*. *Steadfast–1* is the first DT CNN architecture reported to be implemented with only internal Block RAM (BRAM) resources of an FPGA device, which eliminates the memory bandwidth problems as discussed in Section 3.2.1.1. Also note that, the architecture proposed in this thesis is an improvement over *Steadfast–1*.

The topmost block diagram of *Steadfast–1* is given in Figure 3.25. Input of the system is a VGA@60 video stream taken from a PC or a progressive camera, which is captured by a video ADC and directly relayed to an FPGA device in real–time, processed on the FPGA device, and the final result is relayed to a video sink like a PC monitor. Note that, only progressive video streams are supported in order to avoid deinterlacing algorithms, using external memory.

Block diagram of the FPGA implementation is given in Figure 3.26. The *Video Input* block is responsible to acquire the video signal from a VGA standard video interface. The *Video Input* and *Address & Control* blocks are cross–coupled to form a central control unit, i.e., all addresses of the internal buffers and control signals of all sub–blocks are generated by these two blocks. I^2C is also a class of control block, which configures video ADC and DAC units to VGA resolution at power–up. BPU is the first type of processor proposed for *Steadfast–1*, the others being APU(1) and APU($n \geq 2$). BPU calculates (3.1), the part of the cell–state equation whose value is constant through the Euler iterations (g_{ij}), and passes its results to APU(1). APU(1) takes this result, computes

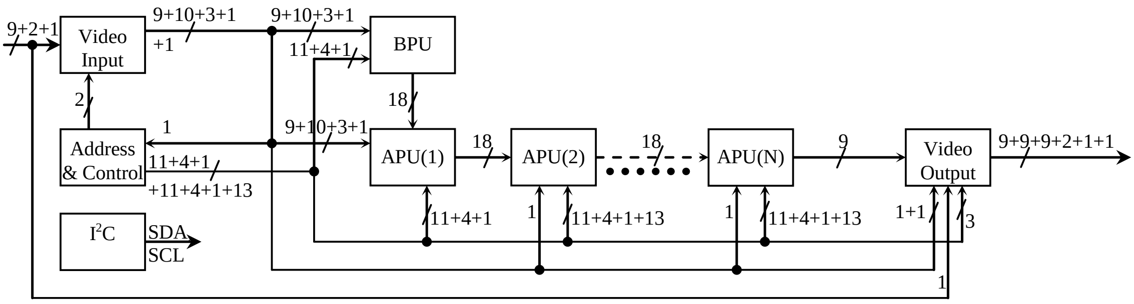


Figure 3.26 Block diagram of the *Steadfast–1* architecture [13]

the first iteration and passes both the first iteration result and its corresponding constant value g_{ij} to APU(2). The second APU calculates the second iteration and passes the result along with the corresponding constant value to the third APU. The processing continues in the same manner until the last processor, APU(N), which produces the final result and passes the value to the *Video Output* block. *Video Output* is responsible to restructure the data as a VGA video stream and generate the control signals required for the standard visual interface.

The internal structure of BPU is given in Figure 3.27. Three lines of the input image are buffered in three BRAM memory units. At the beginning of a new frame, 1st, 2nd and 3rd lines are stored in the 1st, 2nd and 3rd BRAM units, and pixel values are written/read to/from the B and A ports of the BRAM units, respectively. BPU starts processing after capturing the first two lines and third pixel of the 3rd line. The data of the 1st line is not needed for computation after the 3rd line is captured, hence it is overwritten by the data of the 4th line. Every new image line is written over the oldest one and data-handling process continues in this manner. Consequently, BRAM1 stores the 1st, 4th, 7th... lines, BRAM2 stores the 2nd, 5th, 8th... lines and BRAM3 stores the 3rd, 6th, 9th... lines during the process. This memory management structure eliminates the need of moving data between line buffers, but it is more difficult to control.

The coefficients of a $\boldsymbol{B}$ template are held by three DiROM memory units. Each DiROM hold a vertically shifted version of the $\boldsymbol{B}$ template, as given in Table 3.1, to solve some of the complications introduced by the complex line buffering structure. There are three different rotation possibilities of the line indexes stored in BRAM1, BRAM2 and BRAM3:

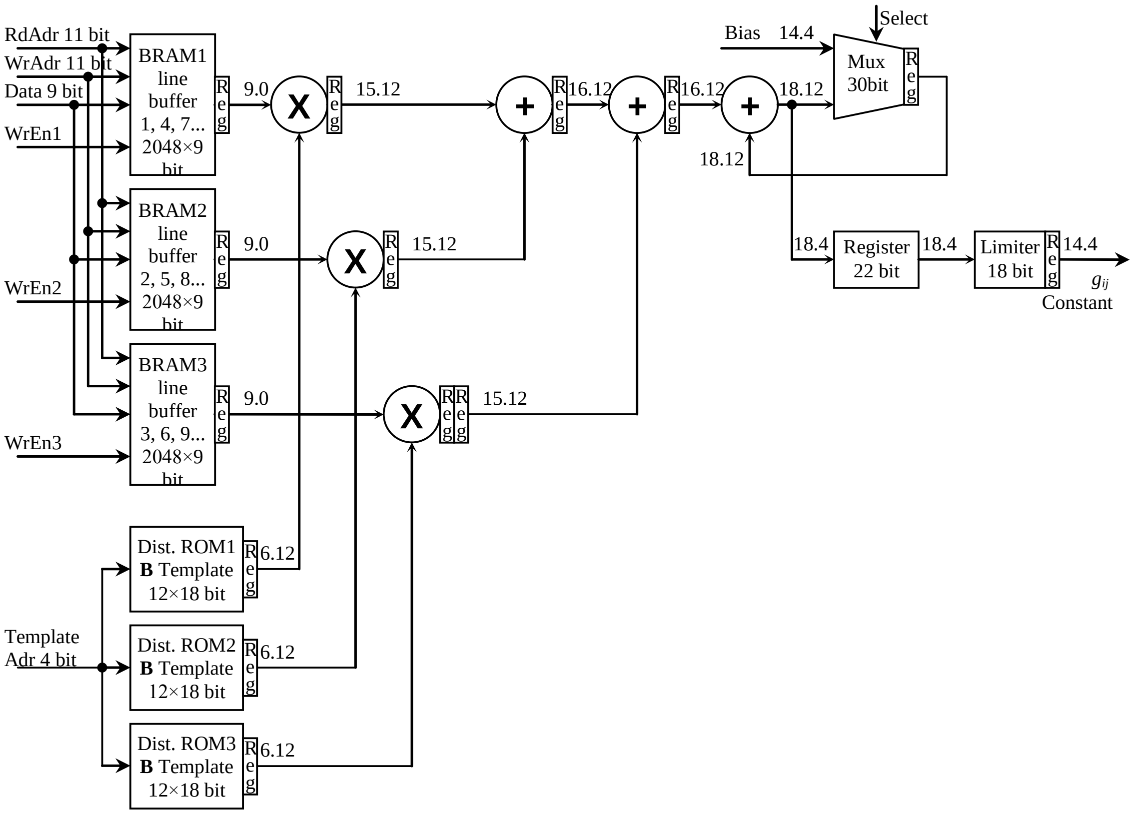


Figure 3.27 Block diagram of BPU of *Steadfast–1* [13]

$i-1$, i and $i+1$; $i+1$, $i-1$ and i; or i, $i-1$ and $i+1$, respectively, as described above. In other words, all possible rotated versions of the template is stored in the template memory to match the line rotations in the BRAM units.

All multipliers and adders are registered to form a pipeline. Three multipliers and three adders are used for nine multiplication and nine addition operations with time division multiplexing, hence the clock rates of all memory units and registers should be three times the pixel rate.

Combining (3.1) and (2.10), the output equation of each iteration is obtained as

$$y_{ij}(n+1) = f\big(\bar{\boldsymbol{A}} \circledast \boldsymbol{Y}_{ij}(n) + g_{ij}\big). \tag{3.2}$$

Table 3.1 Template memory organization of BPU of *Steadfast–1* [13]

Address	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
DiROM1	$b_{-1,-1}$	$b_{-1,0}$	$b_{-1,1}$	—	$b_{1,-1}$	$b_{1,0}$	$b_{1,1}$	—	$b_{0,-1}$	$b_{0,0}$	$b_{0,1}$	—
DiROM2	$b_{0,-1}$	$b_{0,0}$	$b_{0,1}$	—	$b_{-1,-1}$	$b_{-1,0}$	$b_{-1,1}$	—	$b_{1,-1}$	$b_{1,0}$	$b_{1,1}$	—
DiROM3	$b_{1,-1}$	$b_{1,0}$	$b_{1,1}$	—	$b_{0,-1}$	$b_{0,0}$	$b_{0,1}$	—	$b_{-1,-1}$	$b_{-1,0}$	$b_{-1,1}$	—

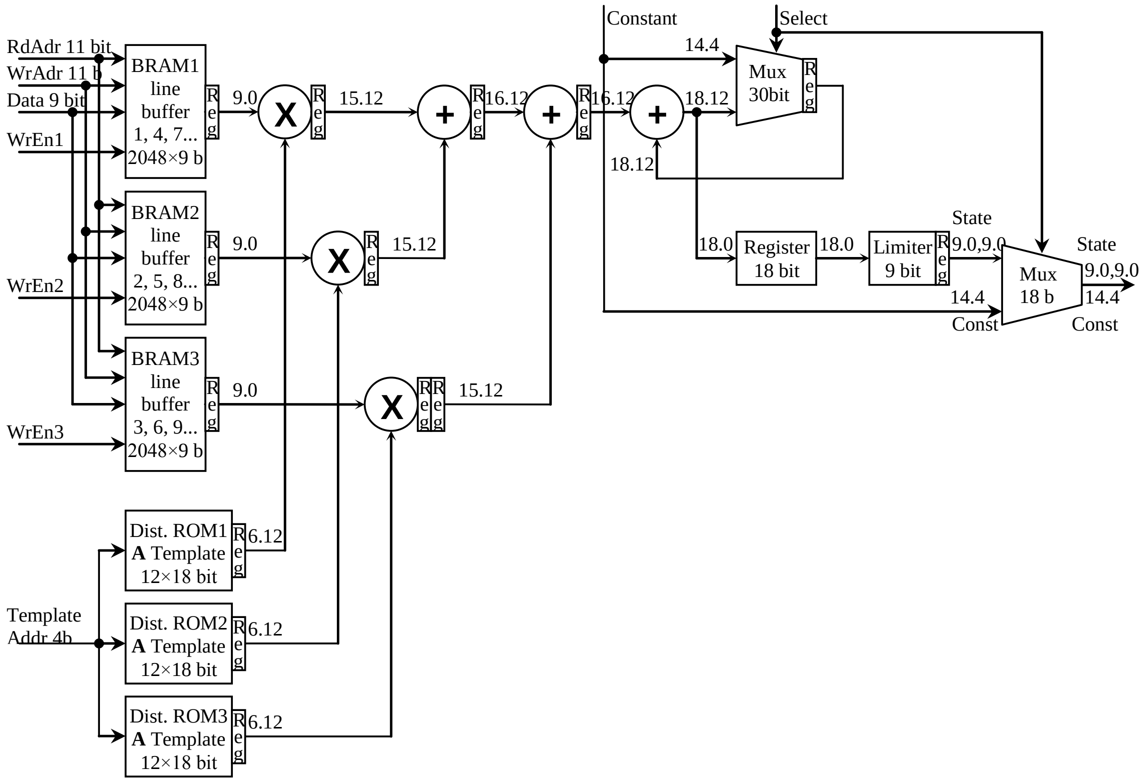


Figure 3.28 Block diagram of APU(1) of *Steadfast–1* [13]

Note that, there are one template-dot-product and one addition operations in both (3.2) and (3.1), hence the structures of an APU and a BPU are fairly similar. Structure of the first APU is given in Figure 3.28. The only difference of the APU(1) from the BPU is an extra output multiplexer, which is used to mix the constant values calculated by the BPU and the first iteration results calculated by the APU(1) to the same bus.

The APU(n) blocks for $n \geq 2$ (APU($n \geq 2$)) are rather complicated compared to the BPU and APU(1) (Figure 3.29). Although arithmetic parts of all processors are the same, memory organization of an APU($n \geq 2$) is completely different from the others: some parts of each BRAM unit is used to store the iteration results relayed from the previous APU, while the other parts are configured to store the BPU results, as BPU result should also be buffered and delayed for synchronization. This complex memory structure is controlled by the central control logic.

A snapshot of the memory management and data flow of the first four consequent APU processors is given in Figure 3.30, where numbers are the line indexes, the numbers enclosed in rectangular frames are the indexes of the lines stored in BRAMs, the numbers

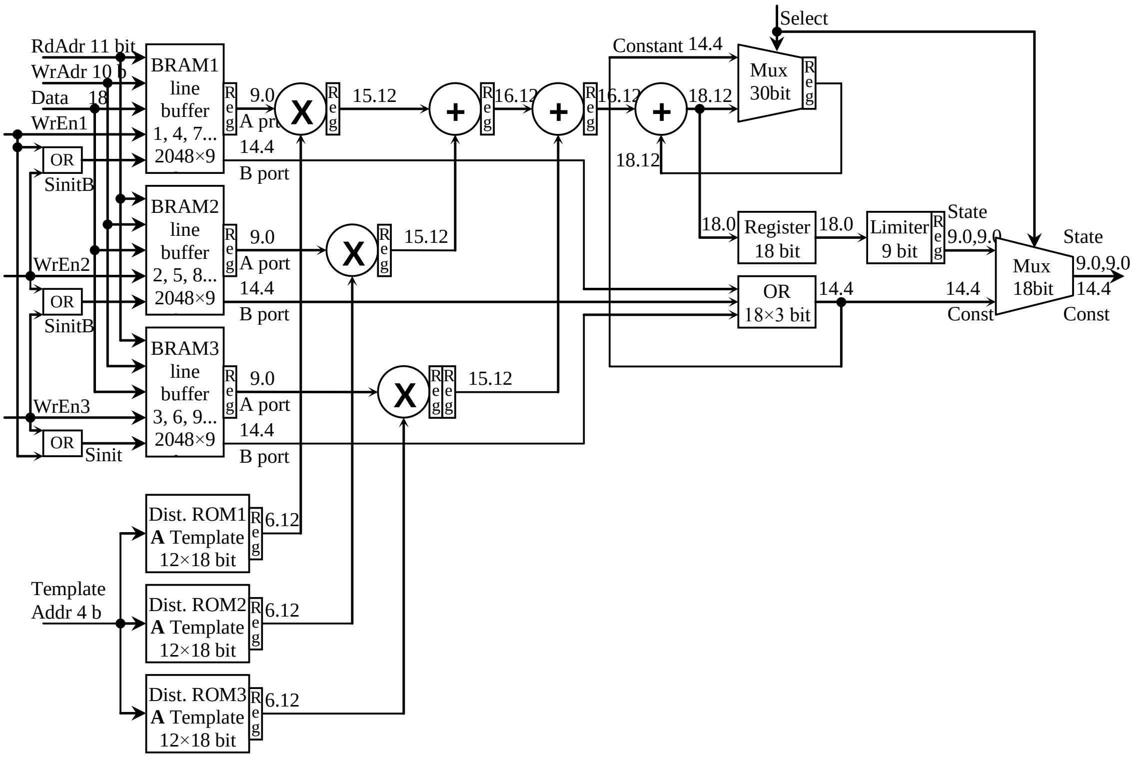


Figure 3.29 Block diagram of an APU(n) for $n \geq 2$ of *Steadfast–1* [13]

enclosed in circles are the indexes of the lines that are currently processed, and the wide arrows show the data transfers between the consequent APU processors. It is seen from the figure that, BRAM line indexes, hence template rotations differ among consequent APU processors, where three different rotations are possible. Rotations of the APU(m) and APU(n) are the same if

$$\mathrm{mod}3(m-n) = 0,$$

e.g., 1st, 4th, 7th; 2nd, 5th, 8th; and 3rd, 6th, 9th APU processors have the same template rotations. Two Most Significant Bits (MSBs) of the A template addresses (DiROM addresses) determine the rotation. Address & Control block produces one of these template address out of three, and the other two addresses are derived from it.

3.3 Conclusion

CT CNN implementations are quite promising at low–resolution focal–plane (near sensor) processing, however, resolution limit of these designs makes them impractical for many modern image processing applications. On the other hand, many DT CNN architectures

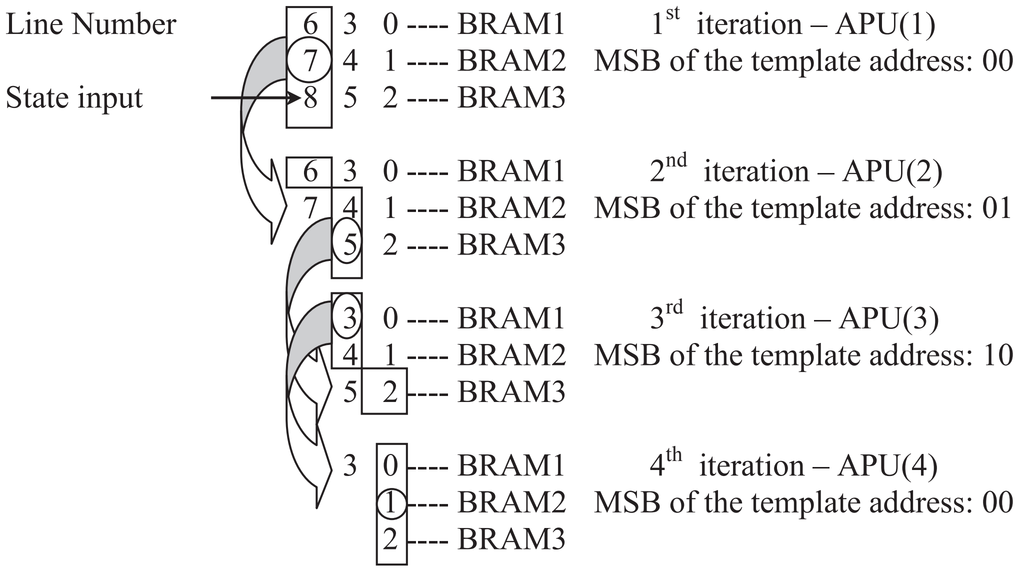


Figure 3.30 Line–flippings of four consequent APU blocks of *Steadfast–1* [13]

are proposed in the literature, but none of them are reported to be capable of processing Full–HD 1080p@60 or faster video signals in real–time. Note that, it can be speculated that it is possible to implement an older design to a Stratix IV FPGA device and achieve a high performance, however, it would require quite an effort and would become obsolete again in a few years. A real solution of this problem is to propose a new architecture with the design considerations of flexibility, modularity and reconfigurability issues.

CHAPTER 4

THE PROPOSED ARCHITECTURE: STEADFAST–2

In this chapter, a flexible, modular and reconfigurable DT CNN architecture is proposed. Although most of the methods, schemes and aspects of the proposed architecture are similar to those discused in Chapter 3, the architecture is completely optimized for the practicality and feasibility of the design, which makes it possible to describe in VHDL and implement on any FPGA device with the highest flexiblity, modularity and reconfigurablity reported, to date.

4.1 Dividing the Computation Process to Multiple Processes

A CNN structure can be simulated or emulated by computing (2.10) on a software or hardware platform, respectively. Considering the high speed aim of this work, it is obvious that a hardware implementation with parallel processing elements is required. However, even if analog structures naturally benefit from parallelization, the converse is true for their digital counterparts. Many difficulties emerge during the design processes of a multi–core digital structure, e.g., controlling multiple input/output signals of different processors, sharing common resources like RAM units, implementing multiple number of iterations, generating boundary conditions for neighboring computation processes, regulating order of the computation, designing a suitable control logic, etc.

Fortunately, cellular structure of CNN is highly regular and continuous, which may be exploited by designing a fully pipelined processor chain. Although pipelined architectures have some latency issues, their throughputs are the same as their input data rates, provided that their input data streams are continuous and regular. In other words, the inability to create a parallel DT CNN structure is compansated by pipelining.

In order to optimize for speed, (2.10) is rewritten as

$$y_{ij}(n+1) = \underbrace{f\left(\bar{\boldsymbol{A}} \circledast \boldsymbol{Y}_{ij}(n) + \overbrace{\bar{\boldsymbol{B}} \circledast \boldsymbol{U}_{ij} + \bar{z}}^{\text{B–Process}}\right)}_{\text{A–Process}} \tag{4.1}$$

where computation is divided to A and B processes. The second template–dot–product operation and addition parts of (4.1)

$$g_{ij} = \bar{\boldsymbol{B}} \circledast \boldsymbol{U}_{ij} + \bar{z} \tag{4.2}$$

is called the *B–process*, which does not depend on the discrete–time variable n. By exploiting this property, it is possible to calculate g_{ij} only once for each input pixel, and use the same result as a constant through all Euler iterations. Now (4.1) can be rewritten as

$$y_{ij}(n+1) = f\left(\bar{\boldsymbol{A}} \circledast \boldsymbol{Y}_{ij}(n) + g_{ij}\right), \tag{4.3}$$

which is similarly called as the *A–Process*.

Considering (4.2) and (4.3), computation flow of the DT CNN can be written as

$$\begin{array}{lcll}
\text{constant calculation} & : & g_{ij} = \bar{\boldsymbol{B}} \circledast \boldsymbol{U}_{ij} + \bar{z} & (4.4a) \\
\text{1st iteration} & : & y_{ij}(1) = f\left(\bar{\boldsymbol{A}} \circledast \boldsymbol{Y}_{ij}(0) + g_{ij}\right) & (4.4b) \\
\text{2nd iteration} & : & y_{ij}(2) = f\left(\bar{\boldsymbol{A}} \circledast \boldsymbol{Y}_{ij}(1) + g_{ij}\right) & (4.4c) \\
\vdots \quad \vdots & : & \vdots \qquad \vdots \qquad \vdots & \\
n\text{th iteration} & : & y_{ij}(n) = f\left(\bar{\boldsymbol{A}} \circledast \boldsymbol{Y}_{ij}(n-1) + g_{ij}\right) & (4.4d) \\
\vdots \quad \vdots & : & \vdots \qquad \vdots \qquad \vdots & \\
N\text{th iteration} & : & y_{ij}(N) = f\left(\bar{\boldsymbol{A}} \circledast \boldsymbol{Y}_{ij}(N-1) + g_{ij}\right) & (4.4e)
\end{array}$$

where N is the total number of Euler iterations desired. *Constant calculation* is an independent process, while any *iteration* depends on the results of the *Constant calculation* and the previous *iteration*. The computation flow is suitable for a fully pipelined architecture, as it is a feed–forward process chain. It is obvious that computation of $y_{ij}(1)$ is not possible before g_{ij} is computed, but it is possible to compute $y_{i-\xi,j-\zeta}(1)$, first iteration result of a previous pixel. Similarly, $y_{i-2\xi,j-2\zeta}(2)$ can be computed while $y_{i-\xi,j-\zeta}(1)$ is still being computed, and so on. The values of the spatial shifts, ξ and ζ, depend on the implementation method, as well as the exact geometry of templates, the input image and blanking areas discussed in Section 4.2.2. Moreover, the uniform spatial shift is distorted at the boundaries, hence it is considerably difficult to give a general mathematical expressions of ξ and ζ. In any case, the local control structure proposed in Section 4.2.3 intrinsically regulates the timings and creates a full–pipeline, completely eliminating the need of calculating the exact values of ξ and ζ.

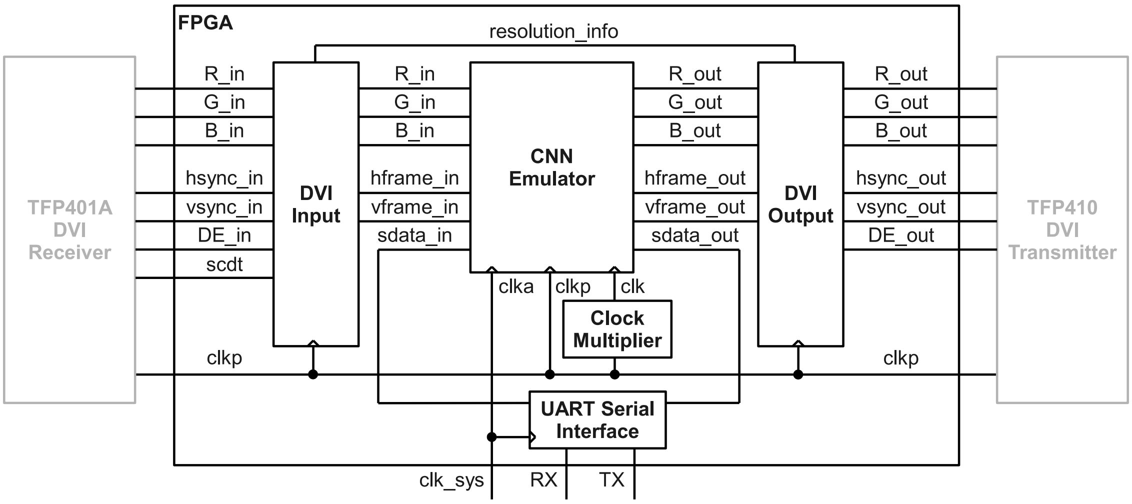
In short, each process given in (4.4) can be assigned to a different processor to form a fully pipelined processor chain, maximizing the speed. Consequently, the number of processors is the same as the number of Euler iterations required for the computation, which depends on the values of the templates, threshold, input image, boundary conditions and initial values of the states.

4.2 Architecture of the Steadfast–2

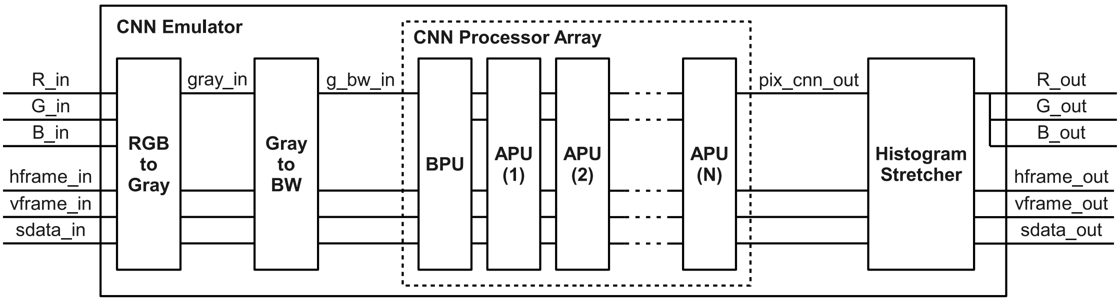
The system is designed to capture a progressive video stream, process it with CNN and convert the result back to a progressive video stream (Figure 4.1a). DVI input and output has been used in prototypes, however, the I/O blocks can be redesigned to support any progressive video stream, such as VGA, DVI, HDMI, DisplayPort, LVDS or any similar progressive video source/sink, or any custom PCI, PCIe, USB or FireWire interface. Interlaced video streams are not accepted as interlacing has small use in modern digital systems, except for broadcasting, not to mention the intensive memory requirements of the deinterlacing algorithms. Similarly, output of the system is also designed to be progressive only.

A block diagram of the FPGA implementation is given in Figure 4.1b. The *Video Input* block is designed to monitor the DVI control signals in order to determine the resolution and frame rate. This block also converts the standard DVI control signals *hsync*, *vsync* and *data enable* to their customized counterparts *hframe* and *vframe*. The generated control signals are passed to the CNN emulator block along with the red, green and blue (RGB)

(a) Block diagram of the system



(b) Block diagram of the *Steadfast–2* implementation



(c) Block diagram of the *CNN Emulator* block

Figure 4.1 Simplified block diagrams of the system, top block of the FPGA implementation and *CNN Emulator* block

pixel data.

Video output block takes the resolution and polarity information from the video input block and use this information to convert custom control signals *hframe* and *vframe* to standard video synchronization signals. These signals are passed to the transmitter hardware along with the CNN emulation results taken from the CNN emulator block.

4.2.1 CNN Emulator Block

A block diagram of the CNN emulator unit is given in Figure 4.1c. The emulation is carried out by a chain of processors. The architecture of the system is designed to be fully pipelined, i.e., in each clock cycle a new input pixel is captured while the computation result of a previous pixel is relayed to the output. Internal buffering structure of each processor eliminates the need of wait cycles. As a result, the system is real–time, and throughput of the system is the same as the input pixel rate.

The first block of the processor chain converts RGB input signal to gray–scale, as almost all CNN applications work on gray–scale or black and white (BW) images. Note that it is also trivial to reconfigure the design for independent RGB color channel processing by using three parallel CNN emulator arrays or use one array and multiplex it between the color channels.

The second block is a programmable gray–scale to black and white (BW) converter, which is implemented using three different methods: thresholding, histogram stretching and CNN. The block is designed to also have a programmable bypass mode to relay the gray–scale input pixel to the next block, which is used when a gray–scale CNN emulation is desired.

The third block is the *B–Processing Unit* (BPU), which calculates (4.4a). Using one BPU is sufficient as B–process is the same for all Euler iterations. The BPU block also passes the original input image $\mathbf{U}$ to the first APU through a second output port, which may be used as the initial value $\mathbf{Y}(0)$.

The fourth block is the first *A–Processing Unit* (APU), which is used to calculate (4.4b). The intermediate constant and initial value matrices, $\mathbf{G}$ and $\mathbf{Y}(0)$, respectively, are used to calculate the first Euler iteration result $\mathbf{Y}(1)$. Also note that the A– and B–processes are very similar as seen from (4.4), except for an $f(\cdot)$ function, which makes it possible to design as single processor which is programmable as APU or BPU.

The first APU is followed by a number of consequent APU blocks, each responsible for calculating one Euler iteration. Hence, the total number of APU blocks is the same as the number of Euler iterations desired. Dividing Euler iterations among many APU blocks, combined with the fully pipelined architecture makes the design comparable to a parallel processing system. Each APU block is also responsible to pass its constant input

G to the next APU, as all APU blocks use the same constant as seen in (4.4b)–(4.4e). Consequently, **G** is stored and used by each APU, and passed through the APU blocks, without being subject to any changes.

Finally, output of the last APU is passed through a programmable contrast stretcher block, which is usually kept in bypass mode, and enabled only to increase contrast of the output image for observation. The block has manual and automatic modes. In the manual mode, constant minimum and maximum pixel values are used for stretching, while these values are calculated dynamically in the automatic mode. The pixel intensities smaller and larger than the minimum and maximum values are saturated to 0 and 255, respectively, while the ones in between are stretched linearly:

$$y_{out} = \begin{cases} 0, & y_{in} < c_{min} \\ 255 * \dfrac{y_{in} - c_{min}}{c_{max} - c_{min}}, & c_{min} < y_{in} < c_{max} \\ 255, & y_{in} > c_{max} \end{cases}$$

where, c_{min} and c_{max} are the minimum and maximum threshold values, respectively. Output of this block is the final emulation result, which is routed to all color channels of the video output block, from which a gray–scale output image is obtained.

4.2.2 Basic Processing Unit

As stated before, (4.2) and (4.3) are very similar, each calculating one template–dot–product and one addition operations, and their only difference is an output function $f(\cdot)$. Consequently, instead of designing two different computation blocks, one programmable basic processing unit called x–Processing Unit (xPU) is designed and used as either BPU or APU (Figure 4.2).

The type of an xPU block can be changed at runtime to BPU or APU by either a hardware or software interface. The APU/$\overline{\text{BPU}}$ port is designed to make the block configurable by another hardware, while a serial programming interface is used to program xPU via an external software running on a PC, microcontroller, etc.

The xPU block has four data I/O ports: data input (data_in, d_{in}), constant input (const_in,

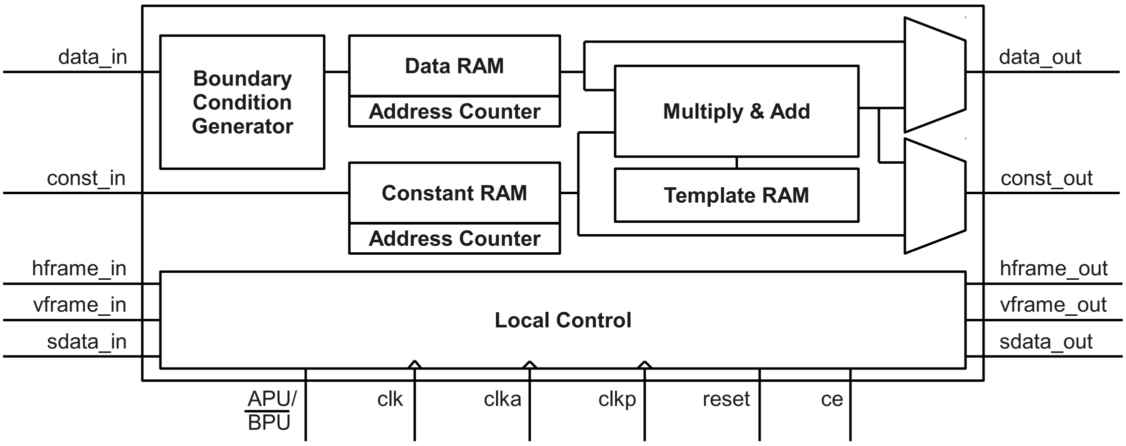


Figure 4.2 Simplified block diagram the xPU

c_{in}), data output (data_out, d_{out}) and constant output (const_out, c_{out}). In the processing work–flow, first, a few lines of the row–wise packed data taken from d_{in} and c_{in} are unpacked in the data and constant RAM units, respectively. Then, template–dot–product operation is carried out between the template and some of the data buffered in data RAM, and then a constant value taken from the constant RAM is added to the result. Finally, the result is multiplexed to outputs, depending on the xPU type, BPU or APU:

$$d_{\text{out}} = \begin{cases} d_{\text{in}} & \text{for BPU,} \\ \bar{\boldsymbol{T}} \circledast \boldsymbol{D}_{\text{in}} + c_{\text{in}} & \text{for APU,} \end{cases}$$

$$c_{\text{out}} = \begin{cases} \bar{\boldsymbol{T}} \circledast \boldsymbol{D}_{\text{in}} + c_{\text{in}} & \text{for BPU,} \\ d_{\text{in}} & \text{for APU.} \end{cases}$$

Here, $\bar{\boldsymbol{T}}$ is a template and $\boldsymbol{D}_{\text{in}}$ is the data corresponding to the template.

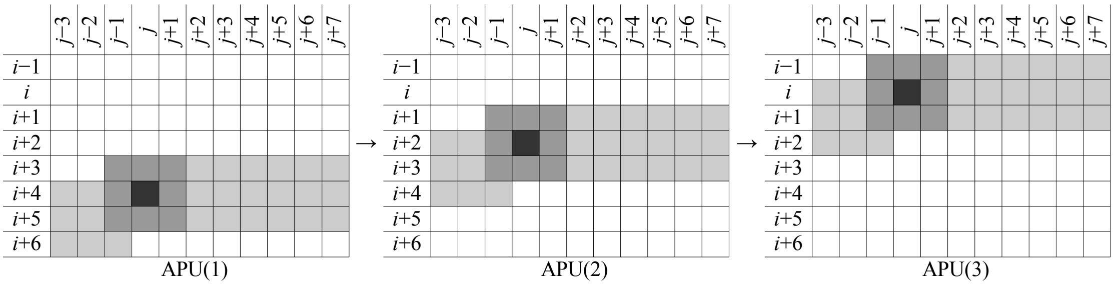
In other words, input image $\mathbf{U}$ is the data of BPU while threshold $\bar{z}$ is its constant. On the other hand, for APU(n), output of the previous APU is its data while output of BPU is its constant.

An xPU has three clock inputs: pixel, processing and auxiliary clocks. The pixel clock is synchronized with the input pixels, as its name implies, while auxiliary clock is used for asynchronous tasks like serial programming. The processing clock is an integer multiple of the pixel clock, which is used to carry out multiplication and addition operations. In other words, arithmetic operations may be carried out using time–division multiplexing, from which the number of multipliers, one of the most valuable resources of an FPGA device, may be reduced. The reduction ratio is equal to the rounded up result of the total number of elements of a template divided by a clock multiplier. For example, considering 3×3 templates, by using processing clocks one, two, three, five and nine times as fast as the pixel clock, the number of multipliers used by an xPU will be $\lceil 9/1 \rceil = 9$, $\lceil 9/2 \rceil = 5$, $\lceil 9/3 \rceil = 3$, $\lceil 9/5 \rceil = 2$ and $\lceil 9/9 \rceil = 1$, respectively. Consequently, the number of multipliers may be reduced by increasing processing clock rate, as long as the resources of the FPGA device used for the implementation supports that clock rate.

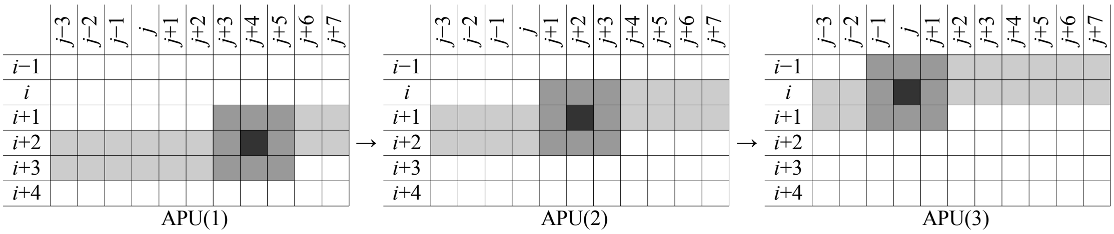
The processor architecture supports non–square templates and does not have any restrictions on the template size, although there are not many ongoing research on either of these topics.

An xPU is programmable to perform either the zero–flux (Neumann) or fixed boundary conditions. The fixed value is also programmable, from which zero boundary condition may be obtained. Toroidal boundary condition is not implemented for resource optimization, as an extra line buffer should be used in each xPU for its implementation. However, note that it is trivial to implement it when needed, as the architecture is designed to be extremely flexible.

template–dot–product operation requires data from $2m+1$ consequent lines of cells to be buffered into a data RAM, which is organized as line buffers [25]. There are two schemes to implement a data RAM: buffering $2m+1$ lines of data directly, or optimizing data RAM and buffering data from $2m$ lines and $2m+1$ cells. In other words, there is no need to store data of a whole line after the optimization. For $m = 1$, a snapshot of the non–optimized and optimized data memory structures of three consecutive APU units are given in Figure 4.3. The non–optimized memory structure is easier to control as the cells at the same column are being processed by different processors. On the other hand, the optimized memory structure consumes less memory in exchange for control complexity, as different control signal should be generated for each processor. In this design, optimized data RAM structure is preferred, whose control difficulties are eliminated by using a local control structure, which was converse in *Steadfast–1*.



(a) Non–optimized memory structure



(b) Optimized memory structure

Figure 4.3 Memory usage of consequent APUs (light gray), and pixels that are being processed (dark gray)

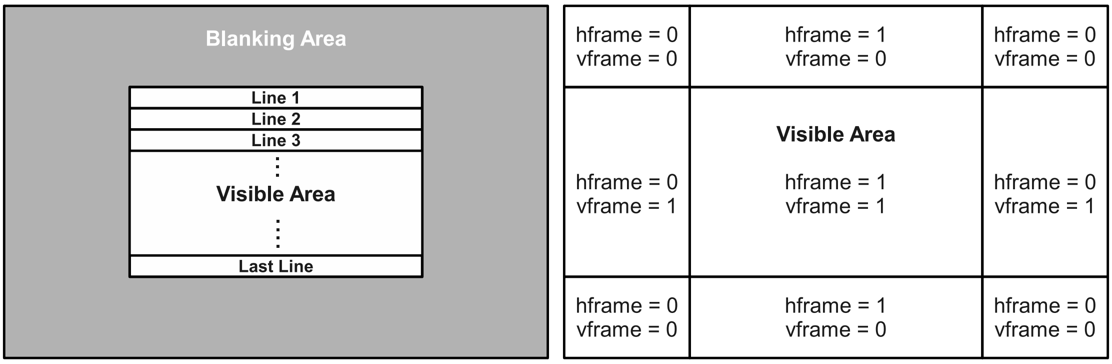
Constant inputs of an xPU should also be buffered in a constant RAM for synchronization. Buffering constants of m cell lines and $m+1$ additional cells is sufficient, as constants are not used in any template–dot–product operation, hence neighboring constants are non required for the computation.

Complexity of the control unit is reduced by assigning a dedicated address counter to each data and constant RAM unit. As a result, it is possible to control these RAM structures with only clock enable and reset signals.

4.2.3 Local Control Structure

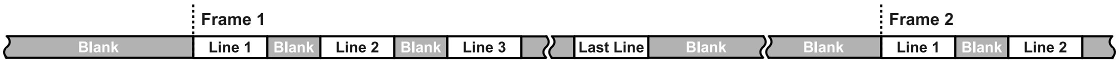
Designing a central control structure is one of the most challenging parts of such a complex design. Furthermore, almost any change or optimization in the design leads to a redesign process of the control logic. The first generation of the proposed structure, *Steadfast–1*, has a complex central control structure, making the design too volatile to improve upon, which was the main reason that a new architecture with local control units is proposed.

In this design, the need of a central control unit is completely eliminated by the introduc-



(a) Visible and blanking areas of a video frame

(b) Video frame segments and the corresponding states of the control signal



(c) Packed video signal

Figure 4.4 Video frame structure defined by video display interfaces and its packing scheme

tion of local control units. All main blocks in the processing chain should have a local control unit, including video input, RGB to gray–scale converter, gray–scale to BW converter, xPU, histogram stretcher and video output blocks. A local control unit basically has two responsibilities: decoding horizontal and vertical frame synchronization signals, *hframe_in* and *vframe_in*, respectively, and generating *hframe_out* and *vframe_out* control signals for the next unit. In other words, each unit is controlled by the previous unit and control the next unit.

Almost all video display interfaces add a blanking area around visible video frames as seen in Figure 4.4a, from which a 1–D video stream is obtained by row–wise packing all 2–D video frames and joining them end to end (Figure 4.4c). Note that, the blanking area causes long and short pauses between frames and lines, respectively.

The video display interfaces also define synchronization signals *hsync* and *vsync*, which act as horizontal and vertical blanking area triggers, respectively. However, pinpointing beginning of a visible area using only *hsync* and *vsync* control signals require a complex control unit, hence digital interfaces like DVI and HDMI also define a visible area indication signal called *data enable*. There is also the signal polarity issue of *hsync* and *vsync*, which may be active high or low, independent from each other, depending only on the video display interface standard.

Instead of being dependent to a video display interface, simplified custom control signals called *hframe* and *vframe* are defined for the implementation. The blanking area given in Figure 4.4a can be segmented into nine regions by extending the edges of the visible area (Figure4.4b). The horizontal control signal *hframe* is defined to be high in the middle of two vertical lines, while vertical control signal *vframe* is defined to be high between two horizontal lines. Consequently, visible area is located when both control signals are high. Moreover, *hframe* and *vframe* control signals identify three regions of the blanking area: left/right, top/bottom and corners. Furthermore, top, bottom, left and right boundaries can also be differentiated by using both the current and one clock period delayed values of *hframe* and *vframe*.

The *Video Input* unit is the first unit of the chain, hence responsible for generating the first *hframe* and *vframe* control signals from video synchronization signals of a standard video display interface. The next block in the chain uses these signals to control its internal operation, and delays these signals the same amount of time of its input to output delay (latency) to control the next block. Note that, delaying for a few lines means delaying for thousands of clock cycles, hence the delaying process is carried out by some pixel and line counters and simple registers instead of huge shift registers in order to use less resources. The same control scheme is carried out by each block in the chain, until *Video Output*, which converts the *hframe* and *vframe* control signals back to the video synchronization signals of a standard visual interface.

4.2.4 Serial Programming Interface

This block is used to program coefficients of the processing units or change their operating modes during runtime. The most challenging part of designing an interface is the need to read/write from/to many sources while preventing data collisions, hence most of the standard serial programming interfaces are either designed to be end to end, or use high–impedance data lines that are not suitable for an FPGA design. Consequently, a packet–based custom communication interface is designed, capable of addressing and

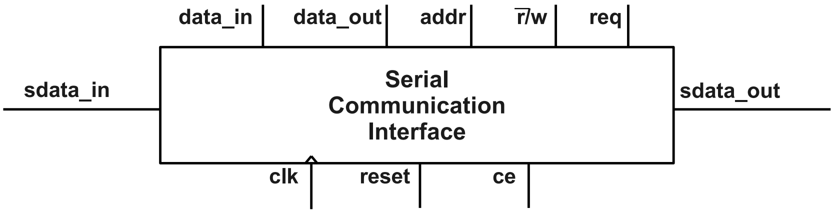


Figure 4.5 Block diagram of a serial communication interface

programming each block in the chain independently.

External serial interface is designed to be RS232 in the prototype, although, any serial communication interface may be implemented by redesigning the *UART Serial Interface* block seen in Figure 4.1b.

Block diagram of a serial communication interface is given in Figure 4.5, which is a type of micro serial transceiver. Each *local control unit* of each block in the processing chain include a serial communication interface. The serial input sdata_out of each block is connected to the sdata_in of the next block to form a seial communication loop inside the FPGA device.

Any coefficients and other configurations of a block in the processing chain is addressed within a memory map as given in Figure 4.6. Up to 2^{11} coefficients and 2^{11} configuration words can be addressed by the serial communication interface. Note that, the word length is designed to be variable to support many different processors in the same chain. For example, the first xPU (BPU), the remaining APU units and the histogram stretching blocks all can be configured with different data word lengths without any conflicts.

The serial data packet carries the following information: a start bit, a processor ID, an internal memory address, a read/write flag, the total length of the message and a data (Figure 4.7). Consequently, first, each processor that needs programming should be assigned with a unique processor ID. Second, all registers that need to be written to or read from the interface should be in the addressable coefficients or Special Functions Register (SFR) space given in Figure 4.6. Note that, a total number of $2^{12} = 4096$ unique blocks can be addressed; and the maximum addressable packet length is $2^5 = 32$, hence a maximum data word length of 224 bits are supported by the interface.

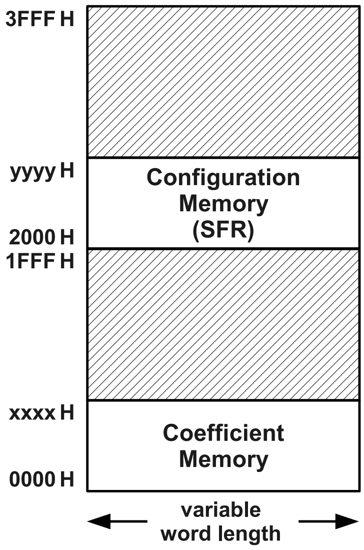


Figure 4.6 Memory map of a block with a serial communication interface

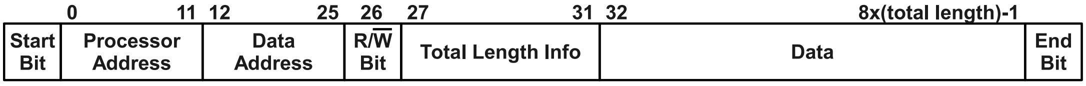


Figure 4.7 Structure of a serial packet

Serial programming work–flow starts with the preparation of a serial data packet by an external programmer. The external data packet is designed to be a second layer data packet, which also includes some redundancies and a checksum. The *UART Serial Interface* block receives this data packet, unpacks the second layer, check the data for corruptions, and relays the first layer data packet to the first processor in the chain. The first processor checks the processor ID, performs the read/write operation if the message is addressed to it, skips otherwise. In any case, the serial packet is transferred to the next processor: if a read operation is performed, the data part of the packet is updated by the processor, else the received data is relayed as it is. Note that, multiple processors of the same type may be assigned the same processor ID and programmed with a single data packet. Each processor in the chain carries out the same operation until the data packet is transferred back to the *UART Serial Interface* block. The UART block repacks the data in a second layer packet and sends it to the external programmer.

4.3 Reconfigurable and Programmable Features of the System

As well known, flexibility, scalability, reusability and practicality issues of any digital hardware design lies within its reconfigurability and programmability properties. The *Steadfast–1* structure proposed in [10] lack these properties, which is one of the main reasons that the improved *Steadfast–2* architecture is proposed. Many features of *Steadfast–2* are designed to be either pre–synthesis configurable or runtime programmable.

4.3.1 Reconfigurable Features

The new architecture is designed to have many parameters that could be configured before the synthesis process, making the design flexible, scalable to smaller or larger FPGA devices and reusable in different implementations with small adjustments. The whole hardware is described in VHDL, and the re–configurable features are implemented by using powerful generics and unconstrained port definitions of VHDL.

The most reconfigurable block of the design is the basic processing unit, xPU. The size of the template, the number of multipliers, the maximum number of visible pixels on a line that will be supported by the processor and fixed–point bit widths of the template coefficients, data ports and constant ports are designed to be pre–synthesis configurable. The whole internal structure of an xPU is automatically generated using these configurations.

Configuring the size of the template of an xPU affects: the number of line buffers of both data and constant RAM units, the number and topology of the the registers used to generate the boundary conditions, and the numbers of multiplication and addition blocks. The processing clock multiplier constant also affect the number of multipliers and adders, along with the total number of entries in a template, from which a multiplication layer and an adder tree are generated automatically during synthesis.

The maximum number of visible pixels that is allowed on a video line is another scaling parameter of an xPU. Assigning a large value to this parameter like 2048 ensures that the processor functions properly for any common resolution, including Full–HD 1080p.

The saturation function at the output of an xPU may be disabled by a generic, which is needed for an ongoing work, realizing a multi–layer DT CNN emulator structure. Details of this structure is beyond the scope of this thesis.

There are some other configurable parameters in the system aside from the ones mentioned for an xPU. The first one of these parameters is the number of xPU blocks in a processing chain. Second, weights of the RGB to gray–scale converter block are designed to have configurable bit widths, although its contribution to resource optimization is insignificant. Finally, each programmable block has assigned a unique ID that is used during serial programming. Other re–configurable parameters are either insignificant, or only used in test/demo settings, hence they also are beyond the scope of this thesis.

4.3.2 Programmable Features

The *Steadfast–1* structure is a static design, requiring a re–synthesis process to change its CNN templates. Therefore, flexibility and practicality of the design are improved almost infinitely by making *Steadfast–2* programmable. Many features of the system are programmable at runtime through a serial inteface, without the need to reconfigure the FPGA device.

The xPU block has many programmable features. First and the most obvious, *template coefficients* and the *threshold constant* may be programmed via a serial programming interface. Second, *xPU type* flag may be programmed as BPU or APU. Note that, threshold value and xPU type information may also be taken externally from the c_{in} and APU/$\overline{\text{BPU}}$ ports, respectively, hence another programmable flag called *xPU mode* sets xPU state to serial or external port programming. Third, *boundary mode* is programmable to fixed or zero–flux boundary conditions. Finally, *fixed boundary value* is also programmable through the same interface.

The gray–scale to BW conversion block is also programmable, as some CNN templates work on gray–scale inputs while others need BW input images. Consequently, the block is programmed to pass either the converted or the original data to its output.

Finally, histogram stretcher block is programmable in three modes: manual, automatic and bypass. In manual mode, minimum and maximum threshold values are also programmed through the serial interface. In automatic mode, histogram of the previous frame is used to calculate the thresholds automatically, where a programmable percentage of data is considered noise at low and high intensities and ignored during the histogram calculation.

CHAPTER 5

IMPLEMENTATION RESULTS AND COMPARISONS

A comparison of the first and second generation *Steadfast* structures is given in Table 5.1. First, the comparison clearly show the flexibility and reconfigurability properties of the new design, as the old structure is stuck to VGA@60 (640 × 480 resolution at 60 Hz frame rate), a single resolution, frame rate and processing clock multiplier. All configurations given in Table 5.1 are tested on an Altera Stratix IV GX 230 FPGA device up to 150 Euler iterations.

Then, the architecture is scaled down to an Altera Cyclone III C 25 FPGA device by reducing the number of iterations to 8 and implementing only table entries with processing frequencies smaller than 300 MHz, which proves that the architecture is highly scalable. It also worth noting that, the actual number of multipliers used by an xPU implemented on a Stratix IV FPGA device should be even, hence one should be added to the odd entries in Table 5.1 for Statix IV devices, which is caused by the dual packed multiplier design of the FPGA device. Consequently, care should be taken when making estimations or comparisons for different FPGA vendors and/or families.

Both prototypes are tested at Full–HD 1080p@60 (1920 × 1080 resolution at 60 Hz frame rate) with a pixel clock frequency of 148.5 MHz and a visible pixel rate of 124.4 Mpix/s, in real–time. On the other hand, maximum number of implemented iterations is 150 for a Stratix IV device, which means that 338.2 giga (338.2×10^9) multiplication and addition operations are carried out per second. Moreover, the pixel rate is limited by DVI I/O chips, hence 2.5 to 3 times higher pixel rates are easily achievable on a Stratix IV FPGA device with a suitable interface, and even higher rates may be achieved with further optimization.

Table 5.1 Resource usage of an xPU for old and new *Steadfast* structures for $m = 1$ (3×3 templates). The numbers at the left and right sides of a '/' are given for Steadfast–1 and 2, respectively, and the symbol '–' is used to indicate 'not implementable'.

					Steadfast–1/Steadfast–2	
Resolution	Frame Rate (Hz)	Pixel Rate (MHz)	Processing Clock Multiplier	Processing Clock Rate (MHz)	Number of 9 Kbit memories	Number of 18×18 multipliers
640 × 480	60	25.175	1	25.175	– / 4	– / 9
			2	50.350	– / 4	– / 5
			3	75.525	6 / 4	3 / 3
			5	125.875	– / 4	– / 2
			9	226.575	– / 4	– / 1
1280 × 720	60	74.250	3	222.750	– / 8	– / 3
			5	371.250	– / 8	– / 2
1920 × 1080	60	148.500	2	297.000	– / 8	– / 5
			3	445.500	– / 8	– / 3

It is also worth to state that, this is the fastest CNN implementation reported to date. Although analog CNN implementations makes the processing faster, video I/O bandwidths and implementable CNN grid sizes are their primary bottlenecks. On the other hand, the I/O bandwidth problem exist for any CNN emulator using external SRAM or DRAM, including any CNN Universal Machine (CNN–UM) implementation. The proposed architecture specifically address the I/O bandwidth problems by introducing a fully pipelined architecture. Furthermore, the structure is designed to be extremely scalable in order to be suitable for any FPGA device, including FPGA devices yet to be developed, hence, performance of the proposed architecture is expected to improve indefinitely with the development of the digital IC technology. Even with the current prototype, the pixel rate ratio of 1080p@60 and VGA@60 is 6.75, which means that the proposed architecture is successfully implemented with a speedup factor of 6.75.

Finally, due to the fully pipelined feed–forward architecture of the design and the proposed local control structure, any number of FPGA devices may be connected end to end to overcome the iteration limit of a single FPGA device. In other words, the design is fully modular as it is, and the workload may be divided to many FPGA devices without any improvements or optimization. Furthermore, the FPGA vendors and/or families are not have to be the same, which is tested by connecting the low–cost and high–end prototypes end to end. Consequently, mixed hardware can be used for applications where multiple FPGA devices should be used, eliminating the need of symmetric hardware extension.

CHAPTER 6

RESULTS AND DISCUSSION

In this thesis a new DT CNN structure is proposed, which is capable of processing full–HD 1080p@60 images in real–time. The design is generically called as RTCNNP–v2 at first, then given the codename Steadfast–2. The proposed architecture is the only DT CNN implementation which is reported to be tested on high–resolution images at a pixel rate of 124.4 Mpix/s, not to mention that the limitation is caused by the I/O bandwidth limit of the DVI daughter cards and not the FPGA implementation itself. Considering the tested processor clock rates up to 450 MHz, it is predicted that a pixel rate about 300 Mpix/s or more can be achieved with a faster video interface, without any changes to the architecture itself.

The original contribution of this thesis is that, the necessity of a global control logic is completely eliminated by designing a new local control structure. In this context, a local control block is embedded in each block in the processing chain, where each block is given the responsibility to control the next block in the chain. In other words, each block gets external control signals from the previous block along with the data to be processed, generate its internal control signals and processes the data, and relay the processed data along with the external control signals generated for the next block. A horizontal and a vertical synchronization signals are used as control signals, which was inspired from standard visual interfaces, but modified to be more suitable for an FPGA implementation.

The local control structure makes the design highly flexible, reconfigurable and reusable, which was proven in a research project, where the proposed infrastructure is used in two more PhD theses, [14] and [16]. In the former, the template–dot–product operator is replaced by a Gabor computation unit, which was also reported in [15]. In the latter, the proposed xPU blocks are configured in different topologies, without being subject to any changes, and a multi–layer CNN emulator has been realized.

The second originality of the proposed architecture is the addition of some programmable features to the design, which was the one of the main missing features of the Steadfast–1 (RTCNNP–v1) architecture. These features make the design practical for an industrial application, as they enable to change the image processing task at runtime. Note that, the originality is not universal, as many other DT CNN implementations are reported to support programmability for a long time.

The third originality is the scalability of the new design, which was proved by implementing the same VHDL source codes on two different Altera FPGA devices: a high–end Stratix IV 230 GX and a low–cost Cyclone III C 25. Note that, none of the Altera primitives or cores are used in the design for portability, hence the same design expected to work on any FPGA device of any vendor. However, also note that, it is not possible yet to imply a PLL in VHDL, hence one exception of the previous statement is the usage of a PLL primitive of Altera. Consequently, the whole design can be transferred to another FPGA vendor or model by simply creating a new project, importing the VHDL source codes, defining a PLL primitive for that FPGA model, defining physical pin locations and recompiling the design.

Finally, some features of a third generation *Steadfast–3* can be proposed for future work. First, the workload of Steadfast–2 is divided to multiple processes only in the time domain, hence it can also be divided in the spatial domain in the next design for more flexibility, which enables processing of 4K@60 or 8K@60 video streams in real–time. Second, even if the main speed advantage of the design comes from not using external memory, designing an external memory controller with limited memory access will widen the application range of the implementation. Third, memory coding technique of the proposed architecture introduces some jitter which limits the maximum frequency or size of the implementation, which can be corrected on a future design. Fifth, computation part of xPU can be modified to support space–variant templates to widen the range of applications. Finally, an optional soft processor core can be implemented on the FPGA device to add some more flexibility to the design. The processor can either manage menial tasks like providing a USB interface, or perform slow processes like database access, decision making, etc.

BIBLIOGRAPHY

[1] Chua, L. and Yang, L., (1988). “Cellular Neural Networks: Theory”, Circuits and Systems, IEEE Transactions on, 35: 1257-1272.

[2] Chua, L. and Roska, T., (2002). Cellular Neural Networks and Visual Computing: Foundations and Applications, Cambridge University Press.

[3] Rodriguez-Vazquez, A., Linan-Cembrano, G., Carranza, L., Roca-Moreno, E., Carmona-Galan, R., Jimenez-Garrido, F., Dominguez-Castro, R. and Meana, S., (2004). “ACE16k: the third generation of mixed-signal SIMD-CNN ACE chips toward VSoCs”, Circuits and Systems I: Regular Papers, IEEE Transactions on, 51: 851-863.

[4] Alba, L., Castro, R. D., Jimenez-Garrido, F., Espejo, S., Morillas, S., Listan, J., Utrera, C., Garcia, A., Pardo, M. D., Romay, R., Mendoza, C., Jimenez, A. and Rodriguez-Vazquez, A., (2009). New Visual Sensors and Processors, vol. 1, Springer Berlin Heidelberg.

[5] Linan, G., Dominguez-Castro, R., Espejo, S. and Rodriguez-Vazquez, A., (2001). “ACE16k: A Programmable Focal Plane Vision Processor with 128 x 128 Resolution”, 2001 European Conference on Circuit Theory and Design (ECCTD), August 2001.

[6] Nagy, Z. and Szolgay, P., (2003). “Configurable Multilayer CNN-UM Emulator on FPGA”, Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on, 50: 774-778.

[7] Malki, S., (2008). On Hardware Implementation of Discrete-Time Cellular Neural Networks, Ph.D. thesis, Lunds University, Sweden.

[8] Martinez, J. J., Toledo, F. J., Fernandez, E. and Ferrandez, J. M., (2008). “A Retinomorphic Architecture Based on Discrete-Time Cellular Neural Networks Using Reconfigurable Computing”, Neurocomputing, 71: 766-775.

[9] Zarandy, A., Keresztes, P., Roska, T. and Szolgay, P., (1998). “CASTLE: an Emulated Digital CNN Architecture Design Issues, New Results”, Electronics, Circuits and Systems, 1998 IEEE International Conference on, September 1998.

[10] Kayaer, K. and Tavsanoglu, V., (2008). “A New Approach to Emulate CNN on FPGAs for Real Time Video Processing”, Cellular Neural Networks and Their Applications, 2008. CNNA 2008. 11th International Workshop on, July 2008.

[11] Yildiz, N., Cesur, E. and Tavsanoglu, V., (2010). “A New Control Structure for the Pipelined CNN Processor Arrays”, Cellular Nanoscale Networks and Their Applications (CNNA), 2010 12th International Workshop on, February 2010.

[12] Cesur, E., Yildiz, N. and Tavsanoglu, V., (2010). “Architecture of The Next Generation Real Time CNN Processor: RTCNNP-v2”, Nonlinear Theory and its Applications (NOLTA), 2010 International Symposium on, September 2010.

[13] Kayaer, K., (2008). Gerçek Zamanlı Video İşleyen Yeni Bir Hücresel Sinir Ağı Emülatörü Tasarımı ve FPGA ile Gerçeklenmesi, PhD Thesis, Yildiz Technical University, Graduate School of Natural and Applied Sciences, İstanbul.

[14] Cesur, E., (2013). Gerçek Zamanlı Bir Hücresel Sinir Ağı Yapısının Tasarımı ve Bu Yapıyla Gabor Filtrelerinin FPGA Üzerinde Gerçeklenmesi, PhD Thesis, Yildiz Technical University, Graduate School of Natural and Applied Sciences, İstanbul.

[15] Cesur, E., Yildiz, N. and Tavsanoglu, V., (2012). “On an Improved FPGA Implementation of CNN-Based Gabor-Type Filters”, Circuits and Systems II: Express Briefs, IEEE Transactions on, 59: 815-819.

[16] Alpay, M., (not finished). Çok Katmanlı Bir Hücresel Sinir Ağı Emülatörünün FPGA Mimarisinin Tasarımı ve Gerçeklemesi, PhD Thesis, Yildiz Technical University, Graduate School of Natural and Applied Sciences, İstanbul.

[17] Yeniceri, R. and Yalcin, M., (2008). “An implementation of 2D locally coupled relaxation oscillators on an FPGA for real-time autowave generation”, Cellular Neural Networks and Their Applications, 2008. CNNA 2008. 11th International Workshop on, July 2008.

[18] Espejo, S., Rodriguez-Vazquez, A., Dominguez-Castro, R. and Carmona, R., (1994). “Convergence and Stability of the FSR CNN Model”, Cellular Neural Networks and their Applications, 1994. CNNA-94., Proceedings of the Third IEEE International Workshop on, December 1994.

[19] Matei, D. and Matei, R., (2008). “Detection of diabetic symptoms in retinal images using analog algorithms”, Proceedings of the 7th WSEAS International Conference on Signal Processing, Robotics and Automation, 2008.Stevens Point, Wisconsin, USA.

[20] Hidvegi, T., Keresztes, P. and Szolgay, P., (2003). “Enhanced Emulated Digital CNN-UM (CASTLE) Arithmetic Cores.”, Journal of Circuits, Systems, and Computers, 12: 711-738.

[21] Nagy, Z., Kiss, A., Zarandy, A., Zsedrovits, T., Vanek, B., Peni, T., Bokor, J. and Roska, T., (2012). “Volume and Power Optimized High-Performance System for UAV Collision Avoidance”, Circuits and Systems (ISCAS), 2012 IEEE International Symposium on, May 2012.

[22] Murari, A., Vega, J., Mazon, D., Arena, P., Craciunescu, T., Gabellieri, L., Gelfusa, M., Pacella, D., Palazzo, S. and Romano, A., (2012). “Latest developments in image processing for the next generation of devices with a view on DEMO”, Fusion Engineering and Design, 87: 2116-2119.

[23] Malki, S. and Spaanenburg, L., (2003). "CNN Image Processing on a Xilinx Virtex-II 6000", Proceedings of the 16th European Conference on Circuit Theory and Design, ECCTD'03, September 2003.

[24] Martinez-Alvarez, J., Toledo-Moreo, J., Garrigos-Guerrero, J. and Ferrandez-Vicente, J., (2010). "A multi-FPGA distributed embedded system for the emulation of Multi-Layer CNNs in real time video applications", Cellular Nanoscale Networks and Their Applications (CNNA), 2010 12th International Workshop on, February 2010.

[25] Lawal, N. and O'Nils, M., (2005). "Embedded FPGA Memory Requirements for Real-Time Video Processing Applications", NORCHIP Conference, November 2005.

CURRICULUM VITAE

PERSONAL INFORMATION

Name Surname	: Nerhun Yıldız
Date of birth and place	: September 21, 1982 – İstanbul
Foreign Languages	: English
E-mail	: nerhun@yahoo.com

EDUCATION

Degree	**Department**	**University**	**Date of Graduation**
Masters	Electronics	Yıldız Technical University	2007
Undergraduate	Electronics and Communications Engineering	Yıldız Technical University	2004
High School	Science–Mathematics	Getronagan Private High School	1999

WORK EXPERIENCE

Year	**Corporation/Institution**	**Job Description**
2005–2012	Yıldız Teknik Üni.	Research Assistant
2004–2005	Entes Elekrtonik Cihazlar İmalat ve Ticaret A.Ş.	R&D Engineer

PUBLICATIONS

Papers

1. Cesur, E., Yildiz, N. and Tavsanoglu, V., (2012). “On an Improved FPGA Implementation of CNN-Based Gabor-Type Filters,” Circuits and Systems II: Express Briefs, IEEE Transactions on, Early Access

Proceedings

1. Yildiz, N., Cesur, E. and Tavsanoglu, V., (2012). “Demonstration of the Second Generation Real-Time Cellular Neural Network Processor: RTCNNP-v2,” 12th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA), aug. 2012

2. Cesur, E., Yildiz, N. and Tavsanoglu, V., (2012). “Demo: An improved FPGA implementation of CNN Gabor-type Filters,” 12th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA), aug. 2012

3. Cesur, E., Yildiz, N. and Tavsanoglu, V., (2011). “An improved FPGA implementation of CNN Gabor-type filters,” IEEE International Symposium on Circuits and Systems (ISCAS), may. 2011

4. Cesur, E., Yildiz, N. and Tavsanoglu, V., (2010). “Architecture of The Next Generation Real Time CNN Processor: RTCNNP-v2,” International Symposium on Nonlinear Theory and its Applications (NOLTA), sept. 2010

5. Yildiz, N., Cesur, E. and Tavsanoglu, V., (2010). “A new control structure for the pipelined CNN processor arrays,” 12th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA), feb. 2010

Projects

1. Scholar, “Durağan Ve Video Görüntü İşleyen Hücresel Sinir Ağı Yapısının Yeni Bir Fpga Mimarisi İle Tasarım Ve Gerçeklemesi,” TÜBİTAK, 108E023, 2009–2011

2. Scholar, “Parmak izi tanıyan hızlı bir sistemin hücresel analog işlemci dizileri kullanarak tasarımı ve uyarlanması,” Yıldız Teknik Üniversitesi BAPK, 25-04-03-01, 2005–2008