Clustering CNN devices for smart networks

Lambert Spaanenburg† and Suleyman Malki‡

† Department of Electronic and Information Technology, Lund University
P.O. Box 118, 22100 Lund, Sweden
‡Department of Electronic and Information Technology, Lund University
P.O. Box 118, 22100 Lund, Sweden
Email: lspaanenburg@ieee.org, Suleyman@eit.lth.se

Abstract– Cellular Neural Networks promise an optimal implementation of non-linear systems. It is hard to keep that promise on platforms with limited capacity. Literature shows that a tiled processor array can be configured to an algorithm-specific processor to facilitate this in a network node. The paper discusses whether such can be scaled up towards smart networks.

1. Introduction

The beauty of Cellular Neural Networks (CNN) lies in the elegant coupling of the mathematical formulation to the silicon implementation. Not every algorithm works nicely on silicon. In fact, the performance gap between function and realization is rapidly growing. This 'more than Moore' gap has become a curse for areas where high speed has to be compromised to low power dissipation. Pervasive computing will bring networks of hundreds of ambient sensors, each as powerful as last year's supercomputer. Clearly this dream can only be fulfilled if the computational power goes down by factors.

The promise of cellular networks is founded on the lack of global control. Where there is no need to schedule / monitor the overall operation, the realization may show up only short wires between synchronous gates and therefore a potentially higher clock rate. But it has not been trivial for CNNs to convert this potential to the real benefit within a large system and/or network.

For historical reasons the older CNN realization is the analog focal-plane image processor [1]. The light diodes capture the image in parallel, but getting the image out by line takes considerable time. At least, it is necessary to separate image extraction from transport by intermediate saving of the pixel values. Notable improvements in performance are further reached by pre-processing the pixels locally to decrease the communication load [2].

The first streaming CNN architectures are based on a pixel-flow mechanism, where a pixel-processing pipeline performs the iterations in time, finally writing the results of a single pixel operation back into memory [3]. The alternative is a tiled architecture, where the streaming is reduced to a pixel line FIFO [4]. Each FIFO element corresponds to a cell in a matrix of simple processing elements with an on-chip network for internal broadcasting. The network applies subsequent instructions while iterating to a stable result before returning to memory.

Recently a CNN processor is developed that can work in both modes. A reconfigurable CNN network is supplemented by program- and data stores, which can be loaded from the attached client processor. After loading, the network can independently perform pixel processing and feature recognition on the current image memory while communicating results with the client processor where higher-order image processing will take place.

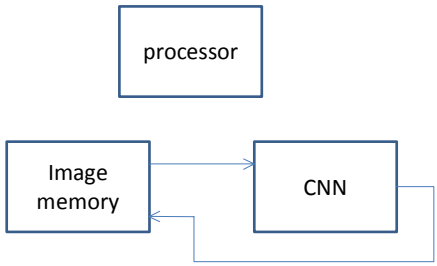

Figure 1: Architecture of streaming CNN system.

The paper is composed as follows. First different scaling aspects of the proposed processor are discussed. Then we look at the modeling of boundary conditions by virtual cells and how these impact network scalability.

2. Scaling aspects of the single CNN node

The CNN equation is in its mathematical notation parametrizable, as the constructs are defined over linear ranges. The nature of the 'more than Moore' gap is essentially this lack of reflection on the potential of the target technology. For a network of devices it is mandatory that all platforms where a function can be executed, this function is executed with either the same or a pre-known performance. In this paper we will largely look at numerical aspects as these set the usability of the results.

Number values are not restricted on the algorithmic level, but will become limited when transferred to the implementation level. Algorithms do not take implementation details such as the effect of finite number representation on a hardware platform into account. The usual design approach starts by building an accurate and precise model, initially cast into software using double-length floating-point numbers for all the values. Realizing this model in a specific technology implies the need to limit the values in accuracy and/or precision.

Accuracy is a concern for realizations where the model parameters cannot be exactly realized or produced by measurements. Numbers may spread out over a value interval, and it must be validated that the algorithm will work even in extreme, i.e. worst-case, situations, the so-called corner cases. Conversely, the realization must be based on numbers in the middle of the assumed value interval, the so-called design centering, and aims to make the design 'robust'. Therefore the proper estimation of *value spread* is a crucial part of the development.

Along similar lines, precision is a concern where the values cannot be realized due to component noise or by limited capacity of the platform. Where accuracy brings the numbers close to the nominal value, precision makes them repeatable. Lack of accuracy may bring values that are still precise but completely off-target, while lack of precision may bring values that are still accurate but change per run of the algorithm. Hence precision needs validation with respect to the 'ideal' model. Conversely, the realization must be based on numbers that are easy for the technology, but that do not lead to *value creep*. This is for instance a typical concern of float-to-fix conversion on a digital platform.

The implementation becomes tractable for systems with constrained inputs where both the value spread and the value creep can be determined by inspecting a finite number of cases. In such circumstances it becomes possible to automatically derive the design constraints for the algorithm in question. A particular test case is the Cellular Neural Network (CNN). It is a regular array of cells, each interacting with the 8 neighbours from the neighbourhood $N_r(c)$ over 8 inputs u and outputs y, both in the range between -1 and +1, according to

$$x^c(k) = \sum_{d \in N_r(c)} a_d^c \, y^d + \sum_{d \in N_r(c)} b_d^c \, u^d + i^c \quad (1)$$

with a final squashing discriminator that turns x into the local u while considering a bias i. All the cells receive 8-bit values, while the coefficients 'a' and 'b' can also be scaled into an 8-bit interval. The worst-case need for internal representation grows gradually to 21 bits (Table 1).

Previous work on CNN template robustness went in two directions. A first point of interest was the formal proof on template universality. As the proof inspects the algorithm, it shows that convergence can exist [5]. The second _ and related _ approach is by adaption of the coefficients to achieve design centering, either algorithmically [6] or statistically [7]. Such research assumes an analogue realization and therefore a fixed precision of 7 – 8 bits [8]. Some effort is on binary coefficients for special applications, such as the pixel snake [9]. The full exploration of robust behaviour is experimentally shown in [10] and gets a more formal basis.

Table 1 Typical data representation of a digital DT-CNN. The notation <k:l> means that the number consists of k-bits integer part and l-bits fractional part.

Value	Fixed-point notation
u and *y* value	<1:7>
a and *b* coefficients	<4:4>
Bias	<5:3>
Multiplication results	<5:11>
State *x*	<10:11>

Fixed-point number representation is usually not enough when the CNN handles dynamic behaviour. It is reported that 36-bits numbers are required to handle 2nd-order differential equations. But this is not necessarily a maximum and one may wonder whether it is enough. Therefore the reasonable extension is the block floating-point format. It splits the requirement for precision and accuracy into two parts: the width of the mantissa sets the precision while the scale factor turns this into accuracy. In case of CNNs, full floating-point representation is not necessary as all the scale factors within the block of nodal template values are identical. Sofar a 5-bit scaling for an 8-bit mantissa has sufficed in experimentation.

3. Networking the nodes into a CNN

As a concept, the Cellular Neural Network promises a 1-cell-per-core solution. But even when the core can be made extremely small, the size of the network will be limited. For a focal-place processor, this is no issue as the network is implicitly at the size of the problem. But in general the problem has to be cut into parts, and in turn the parts have to be suitably mapped onto the platform. As a typical algorithmic approach, the solution is also dependent on the initialization. In the spatial implementation, such initial values are provided over the boundary of the structure. To properly model this in line with the nodal implementation of a cell requires a specialized node to deliver the boundary value. This overhead can be substantial.

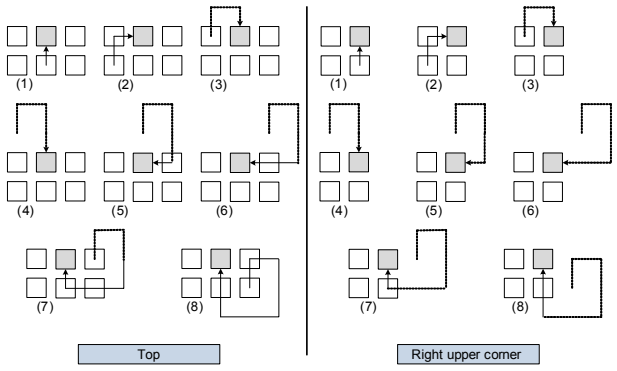

Figure 2: Boundary nodes have an incomplete communication cycle. Squares represent regular nodes while the dotted lines show which part of the packet path is missing. The node of interest is shaded.

Traditionally, the effect of boundary conditions is modelled by adding virtual nodes on the edge of the network. These virtual nodes either supply the boundary nodes with a predefined value (fixed boundary condition) or mirror the value of the boundary node itself (Zero-flux boundary condition). The problem here is further complicated by the asymmetry of the pre-scheduled communication pattern: boundary nodes experience different needs depending on their position in the network. Figure 2 (left) illustrates the disturbed communication cycle for top boundary nodes. The situation is even worse for the corner nodes (Figure 2 right). Actually, not only boundary nodes are affected by the incompleteness of broadcasting but even close-to-boundary nodes as well (Figure 3 left).

Employing the traditional approach of adding virtual nodes is not as simple as it may seem. It is unable to solve the problem entirely and adds on the network size. In any prescheduled communication scheme, virtual nodes should follow the sequence of sending (and eventually forwarding) of values that is accommodated by all regular nodes in the network. This works fine for close-to-boundary nodes (Figure 3 left), but the communication path is still incomplete for boundary nodes. It is clear from Figure 3 (right) that top boundary nodes will not receive any data in steps (4), (5) and (6). In other words, the partially asymmetric transfer cycle necessitates the existence of two (!) layers of virtual nodes to achieve completion. This situation exists for boundary nodes located on the bottom and on the right side of the network as well, but the left edge nodes need only one layer (Figure 2).

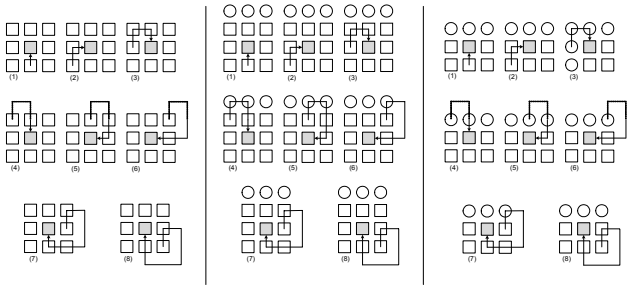

Figure 3: Broadcasting scheme of close-to-boundary nodes (left) is incomplete but the situation is salvaged by adding a single layer of virtual nodes (middle). For boundary nodes more than one layer of virtual nodes is needed (right).

Hence, for an R(ow) x C(olumn) CNN, the number of virtual nodes is equal to 4C+3R+12. Each virtual node needs a router to send and forward packets, a local register and a simplified controller, which will require around 25 slices per node! This seriously degrades area utilization and there is a pressing need to replace the virtual nodes by a simpler mechanism that still completes the communication cycle.

We aim here for a total removal of the need for virtual nodes. This is possible by slightly refining the communication pattern of boundary nodes. Let's consider top and bottom boundary nodes (Figure 2). The actions have to be performed in addition to the regular functionality of the node, mainly when a zero-flux boundary condition is used. For fixed boundary condition most of the sending/forwarding is redundant as all boundary nodes will need to store a single fixed value only that can be used instead of the received value.

Implementing the actions introduces the need for boundary nodes to, sometimes, send or receive two packets simultaneously, which requires a remarkable redesign of the nodal controller and the router in addition to the need of an extra register that keeps one value (W-value). Once again, different boundary nodes will require different refinements. This is of course better than the virtual nodes approach, but still increases the area considerably. A better solution makes use of the existing routing mechanism to forward boundary conditions. We call it 'swing broadcasting' as each boundary node will send its own value to one neighbouring boundary node and then to the other boundary node in the opposite direction. Due to the use of duplex lines between the nodes, the inter-nodal connections have to be idle for one time step in between (Figure 4). In this case, all boundary nodes will have the value of their neighbouring boundary nodes available locally. This requires two additional buffering elements to store the values, but the effect on area utilization is kept at a minimum. Overall, 3 time steps are introduced for each newly calculated y-value.

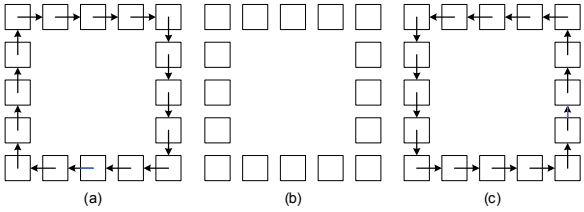

Figure 4: Swing broadcasting allows distributing of boundary conditions in three steps clock-wise (a) and anti-clock wise (c). For proper functionality on the duplex lines a separating idle step is introduced (b).

4. Partition and Merge

But boundary nodes terminate the field of computation. Whether this is good or bad depends on the partition style. In an hierarchical partition each node represents a number of cells. By going down the hierarchy the problem size gets limited and each node represents less cells until eventually we have a 1-to-1 correspondence on the problem sized to the available platform. In a distributive partition we cut through the problem thereby creating boundaries. This essentially disturbs the behavior. So simply slicing is only affordable when bouncing is not required by the interaction of the cells. Most of the image-processing applications are of this type.

Shared memory allows for inter-cell reaction without much change to the boundary concept. If new data is written to a boundary cell, this will automatically be

copied into shared memory. Reading a value from a boundary cell will be re-routed from shared memory, if previously invalidated. Such mechanisms are easily incorporated into the cache coherence protocol of a multi-core system. The system efficiency relies on the designed-in elimination of memory access conflicts. It is illustrative to judge the severity of this problem by looking at the relative size of the boundary. Given n-by-n functional cells we have 4(n+1) boundary cells. The difference n2-4n-4 gets quadratic larger with increasing network size and is already substantial for moderate value. For instance, with n equals 20 we have 5 times more functional than boundary cells.

A shared memory has its main advantages in combination with general-purpose, pipelined cores. Such software-based solutions are known to be much slower than algorithm-specific integrated circuits. Even when both architectures are implemented on an FPGA, the speed difference of a factor 10 is easily incurred. But the use of standard, high-volume parts produced in the latest technology will compensate for this, at least partly.

A modern FPGA is more like a logic-enhanced memory. When a network is partitioned, it creates boundary cells shared 2-by-2 on each cut. Clearly the overall system should work as if no cuts were made, or in other words no boundary cells are inserted. Further it is desirable that the boundary cells do not correspond to package pins.

This seems similar to the Boundary Test approach. In BT we have ports that we want to set, test & scan separate from the chip interior without introducing (many) additional pins. We already have the boundary chain but we do not want the ports to be each physical. In general, the ways to do this are based on multiplexing: (a) in space through a rotating buffer, or (b) in time through encoding.

5. Discussion

The previous methods have enabled the design of a configurable single node CNN architecture. But a single node is not enough to create an intelligent system. Next to interaction, where nodes perform actions based on directions from other nodes, we need reaction whereby also the ongoing actions are changed. The simplest example of such reactivity is where CNN vision nodes adapt the image parts to look at and the granularity to work with.

A typical smart vision sensor is comprised of three layers. On the lowest level we find the pure pixel operations. This is where the need for a high data-handling rate usually leads to a dedicated pixel processor. The feature processing is performed to retrieve information. Finally, this is further condensed to knowledge by reasoning on the extracted features. For example, on the WiCa platform pixel-processing is performed by the IC3D stream-processing chip, while all higher levels are handled by a 80xx processor [11].

The principle benefit of a Cellular Neural Network is that as a computing paradigm it handles both the pixel processing and the feature extraction. Usually it provides also some basic reasoning. As the potential for parallel processing is extended to cover at least information gathering, it is in principle much faster. For the moment, a disadvantage remains that no theory on co-operating CNNs is available.

The hierarchical approach has been used in the development of an algorithm-specific integrated processor that runs simple integer software together with CNN programs [12]. The integer processor is hardly burdened with CNN-related activities but largely functions as network server and knowledge exchanger. This supports the creation of vision through multiple intelligent vision sensors in a low capacity communication network, in extension of arrays of bare cameras with a high-speed connection to a single server.

References

[1] G. Liñán, S. Espejo, R. Dominquez-Castro, E. Roca, and A.Rodriguez-Vázquez, “A 0.5 um CMOS 106 transistors analog programmale array processor for real-time image processing,” Proceedings ESSCIRC, 1999, pp. 358 – 361.

[2] A. Jimenez-Marrufo et al., “Data Matrix Code Recognition Using the Eye-RIS Vision System,” Proceedings ISCAS, 2007, pp. 1214 – 1214.

[3] A. Zarandy et al., “An emulated digital architecture implementing the CNN Universal Machne,” Proceedings CNNA, 1998, pp. 249 – 252.

[4] S. Malki, “On hardware implementation of discrete-time cellular neural networks,” Ph. D. thesis, Lund University (Lund) 2008.

[5] M. Hänggi and G.S. Moschytz. Analytic and VLSI Specific Design of Robust CNN Templates. Journal of VLSI Signal Processing, Vol. 23, 415-427, 1999.

[6] P. Foldesy et alieni. Fault-tolerant design of analogic CNN templates and algorithms-Part I: The binary output case. IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications, Vol. 46, Nr. 2, pp. 312-322, 1999.

[7] S. Xavier-de-Souza et alieni. Towards CNN chip-specific robustness. IEEE Trans. on Circuits and Systems I, Vol. 51, Nr. 5, 892-902, 2004.

[8] B. Mirzai, D. Lím and G. S. Moschytz, “Robust CNN Templates: Theory and Simulation,” Fourth IEEE International Workshop on CNNs and their Applications, Seville, Spain, 1996, pp. 393-398.

[9] V. Brea, M. Laiho and A. Paasio, “Robustness in binary cellular neural networks, Proceedings ISCAS (Kos, Greece, 2006) pp. 2661-2664.

[10] W.-H. Fang, C. Wang and L. Spaanenburg, “In Search for Robust Digital CNN System,” Proc. 10th IEEE Workshop on CNNA and their Applications, Istanbul, Turkey, 2006, pp. 328 – 333.

[11] M.A. Tehrani, R. Kleihorst, P.B.L. Meijer and L. Spaanenburg, “Abnormal Motion Detection in a Real-Time Smart Camera System,” Proceedings ICDSC (Como, September 2009).

[12] L. Spaanenburg, and S. Malki, “Aspects of Algorithm Specific Vision Processors,” to appear in: B. Hoefflinger (ed.), “CHIPS2000 – A guide to our nanoelectronic future,” Kluwer, 2011.