Tree extension of micro-pipelines for mixed synchronous-asynchronous implementation of regional image computations

Valentin Gies and Thierry M. Bernard

ENSTA, 32 Bd Victor, 75015 Paris, France

ABSTRACT

Programmable artificial retinas (PAR) are image sensors with a digital processor in each pixel. PAR based vision systems are our framework. In its basic version, a PAR operates in SIMD mode, which restricts it to low level vision only. To support higher levels of vision, non-SIMD operators have to be settled in each pixel. Programmable connections and asynchronous communications are key ingredients to support regional computations, but under which form ? To identify the good ones, we consider adding pixel data simultaneously over different regions as an exemplary primitive. After recalling previous implementations, we propose a novel one based on a tree extension of micro-pipelines that we call convergent micro-pipelines.

Keywords: asynchronism, micro-pipeline, regional computations, artificial retina, dynamically reconfigurable network

1. INTRODUCTION

An artificial retina is an image sensor with a processing element (PE) in each pixel. Such VLSI circuits are also called "vision chips".[1] Motivated by the low power implementation of vision applications, we focus our research[2] on digital programmable artificial retinas (PAR), for which the PE is a tiny digital processor called the pixellic processor. The latter allows the on-site processing of data from the pixel or its neighbors, according to instructions provided by an external program.

The basic operating mode of a PAR is the SIMD mode (Single Instruction Multiple Data) : at a given time, the same instruction is simultaneously executed by each pixellic processor. SIMD mesh arrays for image processing were popular in the eighties as they allow the efficient implementation of local and shift-invariant operators (linear filtering, mathematical morphology, ...). But they were later abandoned due to several drawbacks. Nowadays, SIMD processing has come back into favor within commercial microprocessors in order to cope with frequency and power consumption limitations. While PARs fully benefit from the SIMD low power advantages, they are much less subject to SIMD drawbacks than the mesh arrays of the eighties. Still, the SIMD mode is only well adapted to low-level vision.

Rather than processed images produced by low-level vision operators, a PAR should ideally output image descriptors, which can only result from higher levels of vision. Therefore, there is a need for non-SIMD resources in the pixel of a PAR. In this paper, we introduce a novel implementation for the regional mixed synchronous-asynchronous computation applied to an exemplary primitive : the regional sum computation. This implementation, based on a tree extension of micro-pipelines that we call convergent micro-pipelines, reduces the implementation cost and replaces dedicated by multi-purpose asynchronous hardware. Furthermore, a peculiar topological exploitation will allow to make the most of it.

2. THE "REGIONAL SUM" AS AN EXEMPLARY NON-SIMD PRIMITIVE

As the visual process goes on, segmentation operators turn the image into a set of regions, each of which is a connected set of pixels. Efficient operators are needed for manipulating regions :

- Inter-regional operators to deal with the topological and metrical properties of the set of regions.
- Intra-regional operators to simultaneously extract features from each region, e.g. moments.

Further author information: (Send correspondence to Valentin Gies)
Valentin Gies: E-mail: contact@vgies.com

Obviously, intra- and inter-regional operators intimately interact during segmentation (e.g. split and merge) and after it.

In contrast with neighbor-to-neighbor communications used in PARs for low-level vision :

- Inter-regional operators need communication between sparse and distant pixels, each representing a different region. This requires a ”segmentation-dependent” communication network over the PAR array.
- Intra-regional operators raise similar issues within each region.

Programmable neighbor-to-neighbor connections[3] allow to implement data-dependant communication networks within the SIMD framework, but with very poor synchronous performances. In the synchronous case :

- The communication speed is limited to ”one pixel farther per clock cycle”.
- Especially for region-to-region communication, only a small proportion of pixels are active at at time, whereas energy is spent to send instructions to all of them.

Suppressing the two above drawbacks leads to use asynchronous instead of synchronous communication. Thus PARs have to feature programmable connections and asynchronous communication to efficiently handle regions. Is it enough ? To carry on the investigation, it is now worth considering the particular but exemplary case of the ”regional sum”. Given an integer number in each pixel, how to simultaneously compute the sum of these integers over each region ? The regional sum is an intra-regional operator. There is no efficient solution to implement it in the SIMD framework. But it seems that programmable connections and asynchronous communications are not enough to help. In the next sections, we first review the few solutions that have been proposed by others to this problem and then describe a novel one, based on the use of micro-pipelines.

2.1. Linear bit-serial multi-input adder

The difficulty for computing a sum with SIMD operators is due to the necessity of moving data. In order to overcome this problem, data must be added locally. To do this, a possibility is to chain pixels with an adder operator inside the pixel. The operator will have to add the binary value provided by the preceding pixel, and the local value. For digital sum computation, bits have to be processed one after the other, from less significant bit to most significant bit. In this case, one also has to sum the carry stored in each pixel during the computation of the preceding bit sum. Finally, the local operator has to be an adder able to compute the sum of 3 binary inputs. The operator used is a full adder.
The principle of the global addition is explained in fig.1

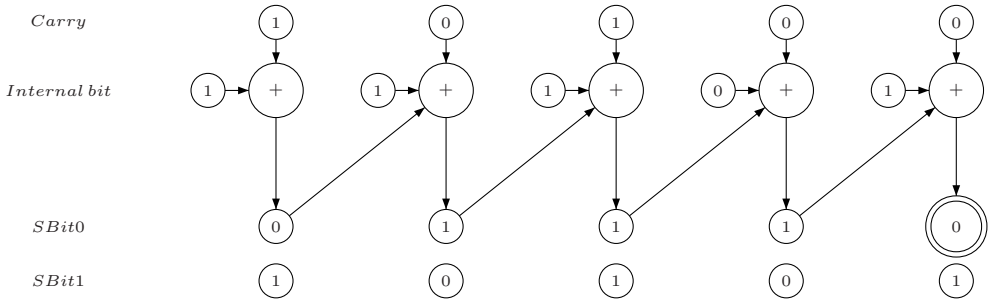


Figure 1. Linear bit-serial multi-input adder process.

This solution has been proposed and implemented by in[4] using dynamically reconfigurable chains of pixels set by external programming.

2.1.1. Sum algorithm

In each processor, the full adder inputs are connected to local binary data (internal bit and carry) and to the preceding full adder less significant output bit (usually called the sum bit). The least significant output bit is connected to the next full adder input in the chain. This bit can also be seen as the parity of the number of 1's input to the full adder in that pixel. By associativity of the parity operator, the result at the end of the chain can be interpreted as the parity of the number of 1's among all local binary inputs in the chain (fig.1). This value is also the least significant bit of the sum of local binary data of the whole region.
To start the regional sum algorithm, the least significant bit of the operand in each pixel is placed in the internal bit while all carries are reset. The first step is to run the global combinatorial sum computation and to get the least significant bit of the sum in the adder at the end of the chain (displayed as a double circled *Sbit0* in fig.1). The second step is to move the most significant bit (called *SBit1* in fig.1) of each full adder in the carry bit. Besides, local values corresponding to the next bit of the operands are loaded in the internal bit of the pixel.
Then, the process is iterated to produce each bit of the sum, as the output of the processor in the pixel at the end of the chain.
This algorithm computes the regional sum in N combinatorial operations where N is the number of bits required to represent the sum. These combinatorial operations are executed in a synchronous sequence.
Since the regional sum operator is based on the ripple propagation, from pixel to pixel of parity information from the beginning to the end of the chain, we consider it as an *asynchronous* operator.

2.1.2. Limitations

Although this implementation allows to compute the regional sum quickly, the main problem is that a chain is a linear structure (cf. fig.2), and it is impossible to cover arbitrary connected sets of pixels with chains. Figure 3 shows a simple example of this impossibility. In such a situation, a tree-based bit-serial multi-input adder is needed instead of a linear one.

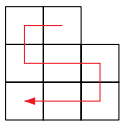


Figure 2. Linear adder

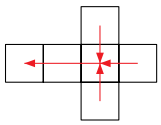


Figure 3. Tree adder

2.2. Tree bit-serial multi-input adder

In this part, we will propose a bit-serial multi-input adder which can be used to compute a sum on an arbitrary region. The region is considered in 4-connectivity, for which a pixel is connected to its four closest neighbors only.
The asynchronous algorithm used to compute the regional sum is an extension of the linear bit-serial multi-input one presented in previous section. The main difference is the tree structure of the global adder.
What is a tree ? It is a direct acyclic graph.[5] That means there is no loop (it is impossible to find 2 points connected by more than one direct subgraph) and there is only one root in it. The acyclic property is needed for using non idempotent associative operators (such as sum) on a graph.[6] The root is be used to collect the sum information computed on the graph. Every pixel in the region is connected to this root through a direct graph (fig. 3). As a consequence, inputs of the adder are connected to local binary data (internal bit and carry) and to *all* the directly preceding full adder least significant output bits in the tree.
For this reason the amount of logic in each pixel depends on the tree-arity. In 4-connectivity, each pixel different from the root can be connected to up to 3 neighbors as input of the adder, the fourth one being necessarily connected to the output of the adder. Taking into account the 2 local binary data, a total of 5 binary inputs are needed for the local adder. A Wallace tree analysis shows that 2 full adders plus one half adder are needed to perform this task in a combinatorial way.
In 8-connectivity, the adder would have been needed 5 full adders and 2 half adders. This solution has been implemented in the Associative Mesh of Orsay[7].[8]
Then, the algorithm used to compute the sum is very similar to the one described in section 2.1.1. The difference lies in the number of inputs of the adders.

At this point, one main issue is still remaining : how can we install a spanning tree over a region using a fast enough procedure regardless of the region shape ?

2.2.1. Asynchronous spanning tree installation

A spanning tree cannot be settled efficiently in a synchronous way, because the number of steps of the algorithm would grow linearly with the geodesic diameter, and it would take a long time. So, we have to perform this task in an asynchronous way.
The asynchronous algorithm used is the following one. At initialisation, all connections between pixels of a same region are established. All pixels are inactives and a root is chosen, deterministically or at random. Then the root is activated, and communicate its state to the neighbor pixels. Each activated pixel keeps in memory the connection through which it was activated, and forward its active state to its neighbors. This process propagates asynchronously throughout the region until all pixels are active. The oriented spanning tree is obtained looking at the unique connections used for the activation of each pixel.
As explained before, a spanning tree is a direct acyclic graph, that means each pixel must have only one antecedent. During the algorithm, a pixel may have to choose between 2 or more antecedents if they want to activate the considered pixel at the same time. For this reason an arbiter is needed in each pixel.

2.2.2. Tree bit-serial multi-input adder implementation in 4-connectivity

The different components needed to perform regional sum computation and spanning tree installation have been defined before. In addition to these functionalities, an SIMD part is present for the synchronisation of the asynchronous phases, for the synchronous local computation and for data storage. This SIMD part has already been largely reported in the literature[92.10] According to the specifications defined, the elementary processor structure of the tree bit-serial multi-input adder is described in fig. 4.

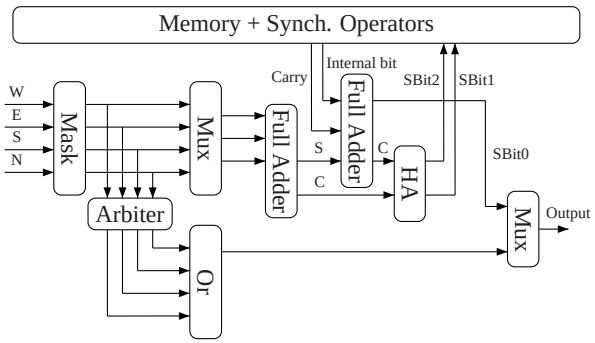


Figure 4. Tree bit-serial multi-input adder processor

2.2.3. Limitations

The tree bit-serial multi-input adder proposed in fig 4 is a generalization of the linear bit-serial multi-input structure. The impossibility of covering arbitrary pixel subsets with chains has been coped with, but the proposed solution has an important hardware cost. The pixel adder has been transformed into a more complicated one, and an arbiter has been added to set up spanning trees (cf. section 2.2.1).
In 4-connectivity, the hardware cost of a 4-input arbiter is about 30 transistors, and 58 transistors for the 5-input adder. In addition, 6 more transistors are needed for the 4-input, 3-output multiplexor used to limit the size of the adder.
In 8-connectivity such as in the Associative Mesh of Orsay,[8] an 8-input arbiter and a 9-input (8 neighbors and an internal bit) adder are used. The arbiter cost is about 120 transistors, and the adder cost (built with 5 full adders and 2 half adders) is about 140 transistors.
Even if these 2 structures are adequate from an algorithmic viewpoint, the number of transistors needed forbids the VLSI integration of a large size PAR.
This limitation would get worse in a N-connectivity network. In such a situation we get the following results :

1. Number of transistors needed for an N inputs arbiter :

$$T_{arbiter} \simeq 14(N-1) + 2N\, log_2(N)$$

2. Number of transistors needed for an N inputs adder :

$$T_{adder} \simeq 24(N - log_2(N+1))$$

Another limitation of the tree bit serial multi-input adder is the use of a dedicated hardware structure for computing regional sums. Such a structure cannot be easily re-used for another computing task. With this approach, every new asynchronous function will require a specific hardware. As a consequence, asynchronous functions will be limited if the number of available transistors in each pixel is bounded, or at the opposite, many asynchronous functions will dramatically increase the hardware cost of each pixel.

2.3. Conclusion

Although the tree bit-serial multi-input adder is algorithmically adequate for regional sum computation, its hardware cost is prohibitive and it can not be use for another task easily. Less expensive and less dedicated alternative would be welcome.

3. CONVERGENT MICRO-PIPELINE ADDER

To overcome the difficulties put in evidence in the previous section, asynchronous operators must be reconsidered in order to reduce them to the strict minimum needed to compute a sum over a spanning tree, and to set up this tree.
Preliminary step of the sum algorithm is the installation of the spanning tree. For this task, the arbiter (which is the operator used to construct the tree, cf. 2.2.1) is necessary and cannot be removed from the elementary processor. Consequently, let's make the most of it.
The arbiter's role is to choose one and only one signal out of the active ones. Only one output of the arbiter can be active. With this behavior, an arbiter can be seen as an automated multiplexor which can select the input corresponding to the first arrived data, and which can change the selected input when the selected data becomes inactive. In order to deactivate the selected input data when there is no need to keep it active, it is necessary to know when this data has been transmitted to the next processor. This functionality corresponds to the control structure of micro-pipelines introduced by I.E. Sutherland[11,12].
Once micro-pipelines adopted to connect neighbor pixels, getting a tree structure requires the ability of merging 2 or more micro-pipelines. Gathering an arbiter and micro-pipelines precisely yield this ability. The corresponding structure is shown on fig.5 and we call it *convergent micro-pipelines*.

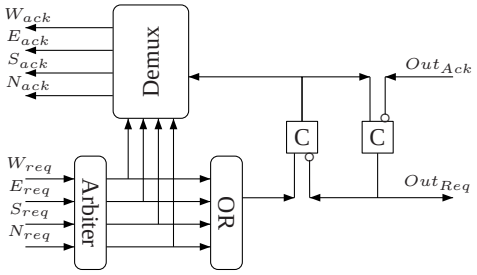


Figure 5. Convergent micro-pipeline structure.

This structure allows to propagate tokens from pixels to pixels through the spanning tree, heading for the root. A token is a set of one or several adjacent pixels where the micro-pipeline is at the logical active state. Unlike the global adder use in previous sections, no operation is performed during the propagation of a token through the tree. However, the number of tokens present in the spanning tree remains constant. The operator just propagates them towards the tree root. For example, if 2 tokens arrive simultaneously at 2 inputs of a convergent micro-pipeline, the arbiter blocks one token during the transmission of the other one. The blocked one is transmitted afterwards.

Since no operations are done during the propagation phase, computations are performed in a synchronous way after the token propagation. This approach reduces the hardware cost since it uses less dedicated operators inside the pixel.
The elementary processor including convergent micro-pipeline is shown in fig. 6. As can be noticed, the hardware structure as been greatly simplified. The hardware cost of the convergent micro-pipeline is about 52 transistors in 4-connectivity, which amounts to a 50% cut in transistor budget compared with the solution of fig. 4. This 50 transistors saving corresponds approximately to the hardware cost of the adder that has been removed. Hardware savings are even more important in 8-connectivity, up to 130 transistors in a pixel.

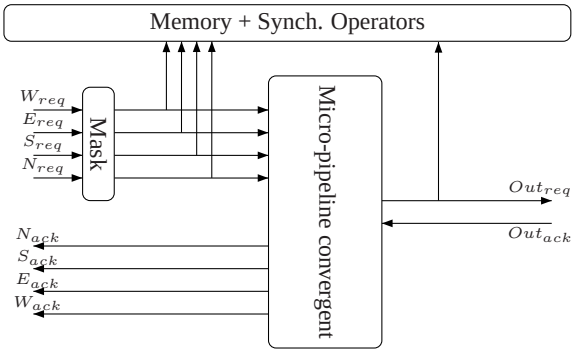


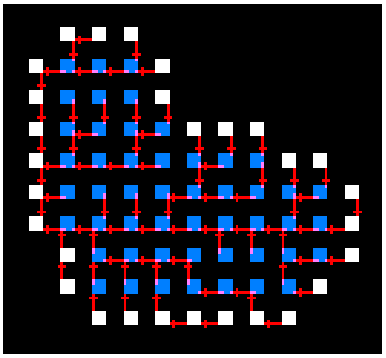
Figure 6. Convergent micro-pipeline based processor

3.1. Region sum algorithm

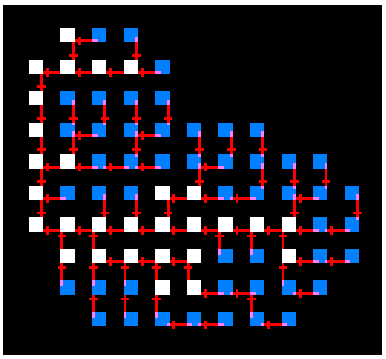
The ability of convergent micro-pipeline operator to gather tokens towards the spanning tree root, provide a way to compute the regional sum without a combinatorial adder in the pixel. First, the algorithm used to set up the spanning tree is the same as the one described in section 2.2.1. As emphasized in section 2.1.1, the least significant bit of a sum of tokens is the parity of this sum. Following this idea, an operation eliminating pairs of adjacent tokens will not change the parity of the set of tokens.
Sum computation is performed with the following algorithm. A token is placed in each of pixel participating to the sum computation (fig.7a). After propagation of these tokens towards the root (fig.7b), pairs of tokens are eliminated by pairs in a synchronous way (fig.7c). This elimination does not depend on tree topology : considering the whole array of pixels as a checkerboard, each white pixel tries to couple its own token with the token located on e.g. its northern black neighbor. That allows an efficient simple pixel pair synchronous SIMD elimination. The tokens remaining after the elimination are then propagated again (fig.7d) and then eliminated by pairs. The process is iterated until there is only 1 or 0 token remaining in the spanning tree. Necessarily, the remaining token will find itself at the root of the tree (fig.7e). This value is the least significant bit of the number of tokens (by eliminating pairs of tokens, the parity of the global number of tokens has not changed).

Actually, pairs of tokens are not eliminated as suggested in the previous paragraph, but replaced by a single token to be used at the next step of the algorithm. Now each step of the algorithm is meant to produce one bit of the sum from LSB to MSB. Then it is meaningful to consider that tokens have a weight. At the first step of the algorithm, the token weight is 1, then each time 2 tokens are coupled, they produce a token of which the weight is twice as large. While computing the i^{th} bit of the sum, the weight of the tokens moving in the spanning tree is 2^i.

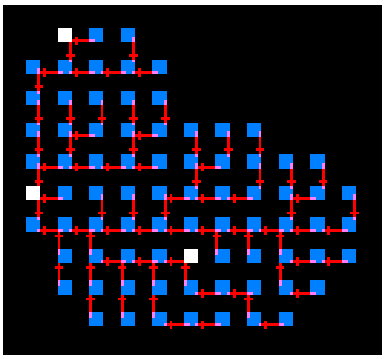
Proposed algorithm allows to compute a sum over a region without using a combinatorial adder in each pixel. This method allows to reduce the hardware cost of the elementary processor. However, the algorithmic cost of this method is increased. Using tree bit-serial multi-input adder (cf. section 2.2), computation of one bit of the global sum is performed with only one propagation through the spanning tree. Instead of that, using convergent micro-pipelines, requires a few propagations (typically 2 to 4). Still, the number of propagation remains linear with the number of bits needed to represent the sum.



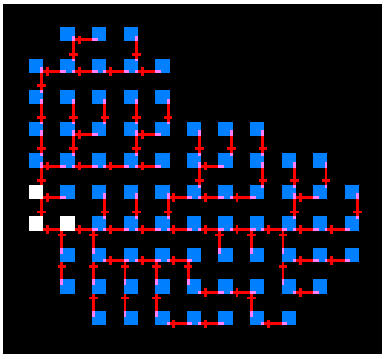
(a) Token initialization



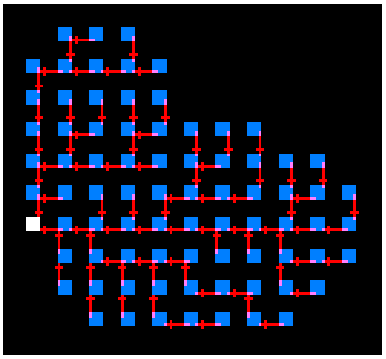
(b) End of tokens propagation



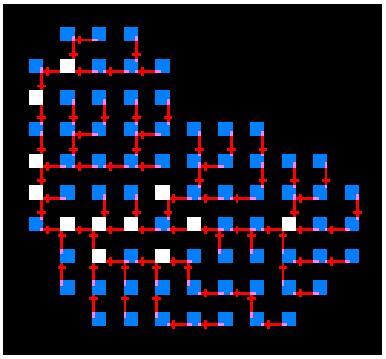
(c) Tokens after pairs eliminations



(d) End of tokens propagation



(e) End of less significant bit computation



(f) 1-weighted tokens loading

Figure 7. Region perimeter computation

3.2. Other algorithmic capabilities

Beyond the reduction of the hardware cost, sum computation is done in a synchronous way, and without using a dedicated adder. Compared with existing solutions, the only hardware structure remaining is the propagation network. Dedicated hardware ressources have been removed from the circuit. Computation operations on the tokens being performed by standard synchronous operators after propagation phases, it is possible to imagine advanced algorithmic capabilities different from binary sum over a region, without adding new hardware.

For example, if elimination is performed on sets of 3 pixels instead of pairs of pixels, one token being generated for each set of 3 pixels deleted, we can compute directly a ternary sum (sum in ternary base) instead of a binary sum. This kind of operation cannot be performed efficiently with processors using a dedicated binary adder such as the tree bit-serial multi-input adder.
Using convergent micro-pipelines allows to overcome some limitations of dedicated functions such as binary adders. An extended survey of the field of possible applications has not been done yet, but as shown in the simple ternary-sum example, such a survey could lead to interesting new regional algorithmic primitives.

4. NETWORK TOPOLOGY FOR REGIONAL SUM COMPUTATION

Hardware cost for implementing regional sum has been significantly reduced by the use of convergent micro-pipelines. However, the use of 4-input convergent micro-pipelines still have an important hardware cost. To cut this transistor expense, a lighter structure is desirable. When looking at an example of computing network, we notice that most of the cells exploits their 4-input convergent micro-pipeline as a simple micro-pipeline only. This is a waste of resources. Is there any way to better distribute the network, that allows the use of k-input convergent micro-pipelines with k smaller than 4 ?
First, let's recall that $k = 1$ corresponds to simple micro-pipeline and is therefore insufficient. What about a 2-input convergent micro-pipeline ? Let's consider a computation network over a region with m pixels. Connecting these m pixels together in a tree structure requires exactly $m - 1$ micro-pipelines, whatever convergent or not. How to settle them among m pixels ? Only one 2-input convergent micro-pipeline in each pixel could be enough. There are 2 main issues for implementing such a structure. The first one is the network topology needed to implement it, the second one is how to install the spanning tree.

4.1. Network topology

We recall that a 4-connectivity network combined with the use of 2-input convergent micro-pipelines is insufficient to set-up a network over an arbitrary shaped region. Let's consider a cross-shaped region of 5 pixels, such a region is an example of this impossibility (Fig. 9). A solution is to increase the connectivity level used. An 8-connectivity topology network could be an obvious solution to the problem, allowing vertical, horizontal and both diagonal connections. However, the hardware cost would be rather expensive. A better solution is to use 6-connectivity and 2-input convergent micro-pipelines. First, let's show that this solution fits our needs. For this, let's consider the pixel matrix as a hexagonal mesh (Fig. 8).
Thanks to hexagonal mesh properties, an arbitrary pixel configuration can be connected with only 2-input convergent

Figure 8. Transformation from square to hexagonal mesh

micro-pipelines. For example, installation of a spanning tree using only 2-input micro-pipelines over a 5 pixels cross-shaped region is proposed (Fig. 9),whereas it was impossible to map in 4-connectivity.

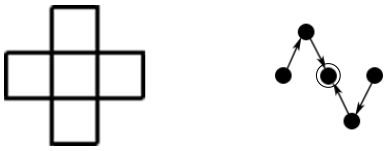


Figure 9. Spanning tree over a cross-shaped region

6-connectivity is the lowest connectivity level allowing the connection of an arbitrary shaped region into a tree structure with only 2-input micro-pipelines, and it leads to the lowest possible hardware cost using convergent micro-pipelines.

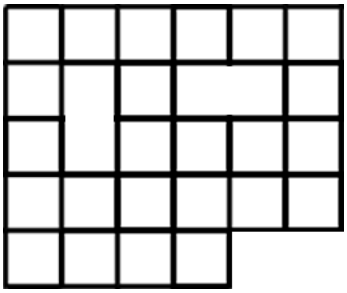
4.2. Asynchronous spanning tree installation

Using 4-input convergent micro-pipelines, installation of a spanning tree is a rather simple task. Starting from a fully connected network, a signal propagates from the tree root through the network until all pixels have been reached (cf. 2.2.1). Using 2-input convergent micro-pipelines, this task is much more difficult because at the initialization of the algorithm, each pixel can be connected to 2 pixels only, and not to all of its neighbors. Connecting all the neighbors is something simplifying the construction of the spanning tree but fortunately it is not really necessary. One as only to ensure that propagation starting from one pixel will reach all the other pixels of the region. That means every pixel of the region has to be connected to all other pixels. Such a region is called a strongly connected component (SCC). The issue is how to build a SCC in 6-connectivity using only 2-input convergent micro-pipelines.

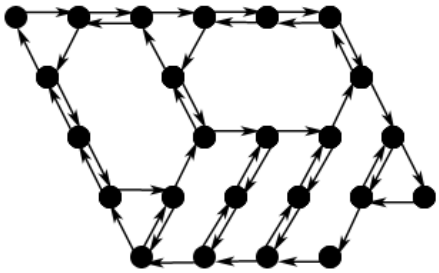
4.2.1. Algorithm principles

A way to solve this problem is to connect all the boundary pixels of the region into a clockwise oriented chain and then to connect all the pixels not connected yet and the boundary rings together. Fig.10 shows the original region at the top, and its corresponding hexagonal representation after the initialization of the connections at the bottom. As explained before, pixel inputs are connected to a maximum of 2 other pixels.

According to this connection method, boundary pixels are connected in a SCC (a ring is a simple SCC), and other pixels



Region being connected



Corresponding hexagonal region
with connections initialized

Figure 10. Example of spanning tree initialization on a 4-connectivity region using a 6-connectivity interconnection network and 2-input micro-pipelines

are added to the SCC thanks to bi-directional connections, which ensures them to be part of the SCC.

The final step of the proposed method is to extract a spanning tree from the SCC by propagating a token from the root as explained before in section 2.2.1. We will not come back on this last point here. A spanning tree corresponding to the SCC proposed in Fig.10 is shown in Fig.11.

4.2.2. Algorithm for connecting a SCC

The algorithm used is very simple, and can be performed in a very cheap and fast synchronous way. Initialization of the connections can be done by only considering 6 local pixel configurations as described in Fig. 12. Configurations 1 to 6

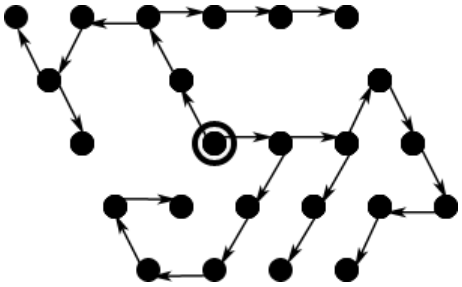


Figure 11. Example of spanning tree obtained on a 4-connectivity region using a 6-connectivity interconnection network and 2-input micro-pipelines

are used for connecting boundary pixels. Configurations 5 and 6 also allow to connect all other pixels diagonally. In the different configurations of Fig. 12, the pixel to connect is double-circled. Black pixels are pixels belonging to the region while white ones are pixels outside the region.
An important fact is the non-isotropy of the local transformation. Configuration 5 and 6 are not rotated versions of configurations 1 and 2 or 3 and 4. This is a consequence of mapping a 4-connectivity square mesh onto an hexagonal network. Instead of configurations 5 and 6, using a $2\pi/3$ rotated versions of configurations 1 and 2 would lead to connect diagonal configurations of pixels not connected in a 4-connectivity squared mesh. Actually, the diagonal connection, not present in 4-connectivity, is used here for establishing all the non-boundary connections.

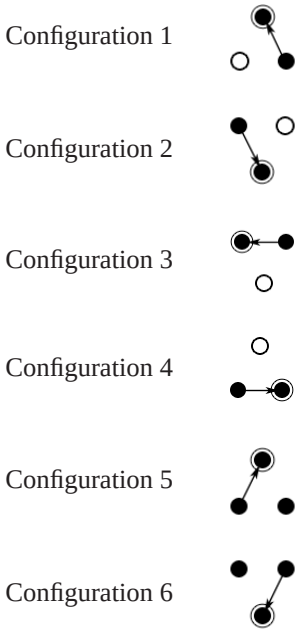


Figure 12. Local configuration for SCC initialization

4.2.3. Validity of the algorithm

Having presented the principles and operation of the algorithm, let's demonstrate its validity. By construction, all pixels of a same region are connected into one SCC. The only point to check is that pixel inputs do not have to be connected to more than 2 neighbor pixels, to allow the use of 2-input convergent micro-pipelines.
As shown in Fig.12, each configuration sets up one input connection. We have to check that if a pixel neighborhood can only match 2 configurations simultaneously. For this let's consider the mutual exclusions of the configurations. Configuration

1 excludes configuration 3 and 5. Configuration 3 excludes 1 and 5, and configuration 5 excludes 1 and 3. Finally, only one odd-numbered configuration can be true at a time. It is the same for even configurations : just mirror odd configurations. Only one even configuration can be true for a given neighborhood. There are no exclusions between odd and even configurations. Finally, a pixel neighborhood can only match one even and one odd configuration. That means a maximum of 2 configurations can be valid at a time, and at most 2-input connections will be set-up in the pixel.

4.2.4. Performance of the algorithm

The proposed algorithm for initializing the SCC can be performed very efficiently in a synchronous non-iterative way. This allows using this algorithm on a massively parallel synchronous machine having only limited resources for synchronous computation. There is no hardware dedicated to the spanning tree initialization task, which means that the reduction of hardware cost due to the use of 2-input convergent micro-pipelines does not imply additional costs for installing the spanning tree. The synchronous hardware used is the same as the one used for local computations such as token elimination (cf. 3.1).

4.3. Hardware reduction

Using 6-connectivity connections over a squared mesh allows to limit the hardware cost of asynchronism to one 2-input convergent micro-pipeline in each pixel plus programmable connections. The cost of a 2-input convergent micro-pipeline is 26 transistors, to be compared with 52 transistors for a 4-input convergent micro-pipeline. However, the required number of programmable connections increases : 2 out of 6 neighbors have to be connected at a time. Consequently, the minimal number of needed programmable connections is 5 connections for each input. Total number of transistors used for programmable connections is now 10 transistors, instead of 4 transistors when using 4-input convergent micro-pipelines. The additional cost is 6 transistors. Finally, 20 transistors are saved in each pixel by using a 6-connectivity topology. This leads to a reduction of 37% of the hardware expense.
The corresponding circuit has been validated by SPICE simulation. Fig.13 shows the convergence of token described in 3. Input tokens arrive by $RI1$ and $RI2$. Output tokens exits through RO.

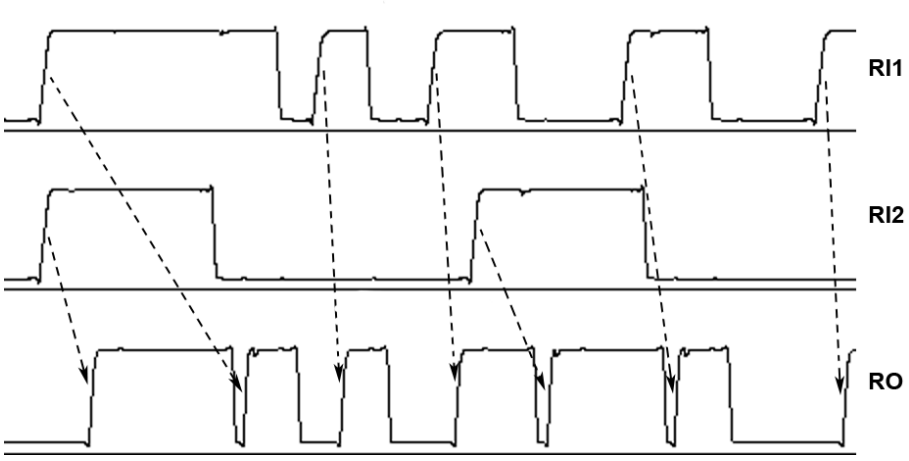


Figure 13. 2 input convergent micro-pipeline oscillogram

5. CONCLUSION

Computing sums of pixel data over regions of an image does not require to have an adder in the elementary processor associated to each pixel. Instead, we propose an alternative approach based on convergent micro-pipelines. This concept allows to reduce the number of transistors in the asynchronous part of the pixel by removing sum dedicated hardware, and provides new algorithmic capabilities. Furthermore, we proposed using 6-connectivity topology and 2-input convergent micro-pipelines for reducing even more the hardware cost of asynchronism. A large scale implementation is now on its

way, and will lead to a retina chip allowing to perform medium level image processing with a wide range of efficient regional operators.

REFERENCES

1. A. Moini, *Vision Chips*, Kluwer Academic Publishers, ISBN: 0-7923-8664-7, 2000.
2. F. Paillet, D. Mercier, and T. Bernard, "Second generation programmable artificial retina," in *IEEE ASIC/SOC Conf.*, pp. 304–309, Sept. 1999.
3. H. Li and Q. Stout, *Reconfigurable Massively Parallel Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
4. T. Komuro, S. Kagami, and M. Ishikawa, "A dynamically reconfigurable simd processor for a vision chip," *IEEE Journal of Solid-State Circuits* **39**(1), pp. 265–268, 2004.
5. B. Ducourthial and A. Merigot, "Graph embedding in the associative mesh model," Tech. Rep. TR-96-02, 1996.
6. B. Ducourthial and A. Mérigot, "Parallel asynchronous computations for image analysis," *Proceedings of the IEEE* **90**(7), pp. 1218–1228, 2002.
7. A. Merigot, "Associative nets: A graph-based parallel computing net," *IEEE Transactions on Computers* **46**(5), pp. 558–571, 1997.
8. D. Dulac, S. Mohammadi, and A. Merigot, "Implementation and evaluation of a parallel architecture using asynchronous communications," in *CAMP*, pp. 106–111, 1995.
9. A. Manzanera, *Vision artificielle rétinienne*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 2000.
10. A. Manzanera, "Morphological segmentation on the programmable retina: towards mixed synchronous/asynchronous algorithms," in *ACM ISMM Conf.*, pp. 389–399, Apr. 2002.
11. I. Sutherland, "Micropipelines," *Communications of the ACM* **32**(6), pp. 720–738, 1989.
12. S. Hauck, "Asynchronous design methodologies: An overview," Tech. Rep. TR-93-05-07, 1993.