Correctness of the FPNA neural paradigm

Bernard Girau

Correctness of the FPNA neural paradigm

Bernard Girau

Abstract

Neural networks are usually considered as naturally parallel computing models. But the number of operators and the complex connection graphs of standard neural models can not be handled by digital hardware devices. A new theoretical and practical framework allows to reconcile simple hardware topologies with complex neural architectures: *Field Programmable Neural Arrays* (FPNA) lead to powerful neural architectures that are easy to map onto digital hardware, thanks to a simplified topology and an original data exchange scheme. This report describes the basic principles of the FPNA paradigm. Formal definitions are introduced and illustrated. Two computation methods for feedforward FPNAs are introduced. The proof of their correctness is sketched.

Note: FPNAs have already been studied in LORIA report [Gir99c]. This report is based on a slightly simplified FPNA definition, so as to focus on the correctness problem which has not been studied in [Gir99c].

1 Introduction

Various fast parallel implementations of neural networks have been developed ([NS92]). The very fine-grain parallelism of neural networks uses many information exchanges, so that it better fits hardware implementations. Configurable hardware devices such as FPGAs (Field Programmable Gate Arrays) offer a compromise between the hardware efficiency of ASICs and the flexibility of a simple software-like handling. Several works show that FPGAs are a real opportunity for flexible hardware implementations of neural networks ([Gir00]). And yet the implementation of standard neural models raises specific problems.

Many neural implementations on FPGAs handle simplified neural computations ([Gir00]). Moreover, many efficient implementation methods (ASIC, neuro-computer, ... [Mor95]) have to limit themselves to few well-fitted neural architectures. An upstream work is preferable: neural computation paradigms may be defined to counterbalance the main implementation problems, and the use of such paradigms naturally leads to neural models that are more tolerant of hardware constraints, without any additional limitation. Since the main implementation difficulties are linked to area-greedy operators and complex topologies, two kinds of *hardware-adapted neural computation paradigms* may be found. Several models, such as bit-stream neural networks ([Sal94, vDJST93]), allow to handle area-saving neural computations, whereas the work of [Gir99b] leads to complex neural processings based on simplified topologies.

The *Field Programmable Neural Arrays* of [Gir99b] are based on a FPGA-like approach: a set of resources whose interactions are freely configurable. These resources (links and neural operators) are defined so as to perform computations of standard neurons, but they behave in an *autonomous* way. As a consequence, numerous *virtual* links may be achieved thanks to the application of a multicast data exchange protocol to the resources of a sparse neural network. This new neural computation concept enables a simplified neural architecture to replace a virtual complex one.

The practical study of [Gir99b, Gir99d] shows that FPNAs lead to very efficient hardware implementations of neural networks. The theoretical study of FPNAs ([Gir99b, Gir99a]) ranges from the definition of computation methods for each class of FPNAs to the determination of their computation power. This report focuses on the computation of feedforward FPNAs. Section 2 first defines these neural models. Then it introduces a sequential form and a parallel form of their computation. Section 3 illustrates these notions with a simple example. Section 4 finally aims at justifying the correctness of the computation algorithms of section 2.

2 FPNAs

Two kinds of autonomous neural resources appear in a FPNA: *neurons* that apply standard neural functions to a set of input values on one hand, and communication *links* that behave as independent affine operators on the other hand. In a standard neural model, each communication link is a connection between the output of a neuron and an input of another neuron. The number of inputs of each neuron is its fan-in in the connection graph. On the contrary, communication links and neurons become *autonomous* in a FPNA: their dependencies are freely programmable.

More precisely, the communication links connect the nodes of a directed graph, each node contains one neuron. The specificity of FPNAs is that relations between *any* of the local resources of each node may be freely set. A link may be connected or not to the local neuron *and to the other local links*. Therefore direct connections between affine links appear, so that the FPNA may compute numerous composite affine transforms. These compositions create numerous *virtual neural links*.

2.1 Formal definition of FPNAs

A FPNA is defined[1] by means of:

- a directed graph $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is a finite set of nodes, and $\mathcal{E}$ is a set of directed edges without loop: for each node n, the set of the direct predecessors (resp. successors) of n is defined by $Pred(n) = \{p \in \mathcal{N} \mid (p,n) \in \mathcal{E}\}$ (resp. $Succ(n) = \{s \in \mathcal{N} \mid (n,s) \in \mathcal{E}\}$), the set of the input nodes is $\mathcal{N}_i = \{n \in \mathcal{N} \mid Pred(n) = \emptyset\}$,
- a set of neurons $((\theta_n, i_n, f_n))_{n \in \mathcal{N}}$, where $\theta_n \in \mathbb{R}$, i_n is a function from $\mathbb{R}^2$ to $\mathbb{R}$, and f_n is a function from $\mathbb{R}$ to $\mathbb{R}$: for each node there is one neuron resource that sequentially handles any neuron computation[2],
- a set of affine functions $(x \mapsto W_n(p)x + T_n(p))_{(p,n) \in \mathcal{E}}$: for each node there are as many communication links as this node has got predecessors, each communication link is associated with an affine operator,
- for each node n in $\mathcal{N} - \mathcal{N}_i$,
  - a positive integer a_n: number of iterations before a neuron applies its transfer function,
  - for each p in $Pred(n)$, a binary value $r_n(p)$: set to 1 iff the link (p,n) and the neuron in n are connected,

[1] This report presents a simplified FPNA definition which is sufficient as long as hardware implementations are not taken into account.

[2] Any standard neuron computation may be performed by means of a loop that updates a variable with respect to the neuron inputs, and a final computation that maps this variable to the neuron output. θ_n stands for the initialization value (see [Gir99b]). The iteration function i_n stands for the updating function inside the loop. The neuron output is finally computed with f_n.

- for each s in $Succ(n)$, a binary value $S_n(s)$: set to 1 iff the neuron in n and the link (n,s) are connected,
- for each p in $Pred(n)$ and each s in $Succ(n)$, a binary value $R_n(p,s)$: set to 1 iff the links (p,n) and (n,s) are connected,

- for each input node n in $\mathcal{N}_i$,
  - a positive integer c_n: number of global inputs sent by this node,
  - for each s in $Succ(n)$, a binary value $S_n(s)$ (see above).

2.2 Computing in a feedforward FPNA

2.2.1 Feedforward FPNAs

A FPNA is feedforward if the local configured connections between resources do not infer any cyclical dependency. Therefore a FPNA may be feedforward even if its graph $(\mathcal{N},\mathcal{E})$ is cyclical. Two formal definitions express this (where (n,n) stands for the neuron resource in node n).

Definition 1 *Let $\phi = \left\{\mathcal{N},\mathcal{E},(\theta_n,i_n,f_n,W_n,T_n,a_n,R_n,r_n,S_n)_{n\in\mathcal{N}}\right\}$ be a FPNA. Its dependency graph $\mathcal{G}_D(\phi) = (\mathcal{N}',\mathcal{E}')$ is defined by:*

- $\mathcal{N}' = \mathcal{E} \cup \{(n,n) \mid n \in \mathcal{N}\}$
- $\mathcal{E}' = \{((p,n),(n,s)) \mid (R_n(p))(s) = 1\} \cup \{((p,n),(n,n)) \mid r_n(p) = 1\}$
  $\cup \{((n,n),(n,s)) \mid S_n(s) = 1\}$

Definition 2 *Let ϕ be a FPNA. It is feedforward iff $\mathcal{G}_D(\phi)$ does not include any cycle.*

2.2.2 Computation principles

Several computation methods have been defined for FPNAs. In any such method, all resources behave independently, and when a resource receives values, it applies its local operator(s), and sends the result to all neighboring resources to which it is locally connected (a neuron resource waits for a_n values before sending any result to its neighbors). Unlike standard neural computations, the FPNA paradigm allows a resource to be connected or not to a neighboring resource. Moreover, a communication link may handle several values, and it may directly send them to other links.

Two main computation methods have been defined for feedforward FPNAs. In each one, c_n values $\left(x_n^{(i)}\right)_{i=1..c_n}$ are given for each input node $n \in \mathcal{N}_i$ (global inputs of the FPNA). Moreover each node n in $\mathcal{N} - \mathcal{N}_i$ has got local variables c_n and x_n, initially set as $c_n = 0$ and $x_n = \theta_n$.

2.2.3 Asynchronous sequential computation

This computation method handles a list of tasks $\mathcal{L}$ that are processed according to a FIFO scheduling. Each task $[(p,n),x]$ corresponds to a value x sent on a communication link (p,n).

Initialization: For each input node $n \in \mathcal{N}_i$, c_n tasks $[(n,s),x_n^{(i)}]$ are created for all s in $Succ(n)$ such that $S_n(s) = 1$. The order of creation corresponds to a lexicographical order on (n,i,s) (with respect to the order of $\mathcal{N}$).

Sequential processing: *(while $\mathcal{L}$ is not empty)*

Let $[(p,n),x]$ be the first element in $\mathcal{L}$ (immediately removed from $\mathcal{L}$)

1. *compute* $x' = W_n(p)x + T_n(p)$
2. *for all* $s \in Succ(n)$ *such that* $(R_n(p))(s) = 1$, *create* $[(n,s),x']$ *according to the order on* s
3. *if* $r_n(p) = 1$ *(the neuron in* n *is said to be receiving the value of task* $[(p,n),x]$*)*
   - *update* c_n *and* x_n *:* $c_n = c_n + 1$, $x_n = i_n(x_n, x')$
   - *if* $c_n = a_n$ *(the local neuron computes its output)*
     (a) $y = f_n(x_n)$, $c_n = 0$, $x_n = \theta_n$
     (b) *for all* $s \in Succ(n)$ *such that* $S_n(s) = 1$, *create* $[(n,s),y]$ *according to the order on* s

2.2.4 Asynchronous parallel computation

The above sequential computation may be seen as the reference computation algorithm for FPNAs. A parallel version has been defined so as to fit parallel digital hardware implementations. A request-acknowledge protocol must be handled. A request $req[(p,n),(n,s),x]$ corresponds to a value x sent by resource (p,n) to resource (n,s).

Initialization: For each input node $n \in \mathcal{N}_i$, c_n requests $req[(n,n),(n,s),x_n^{(i)}]$ are created for all s in $Succ(n)$ such that $S_n(s) = 1$.

Concurrent processing: All resources in parallel sequentially handle all the requests that they receive. Resource (n_1,n_2) chooses $req[(n_0,n_1),(n_1,n_2),x]$ among the unprocessed requests already sent to (n_1,n_2) with a fair policy, and then it processes it:

1. acknowledgement for (n_0,n_1)
2. if $n_1 = n_2$ then
   - $c_{n_1} = c_{n_1} + 1$, $x_{n_1} = i_{n_1}(x_{n_1}, x)$
   - if $c_{n_1} = a_{n_1}$ then
     - for all s in $Succ(n_1)$ so that $S_{n_1}(s) = 1$, create $req[(n_1,n_1),(n_1,s),f_{n_1}(x_{n_1})]$
     - reset : $c_{n_1} = 0 \quad x_{n_1} = \theta_{n_1}$, wait all acknowledgements

   else
   - $\forall\, s \in Succ(n_2)$ s.t. $R_{n_2}(n_1,s) = 1$, create $req[(n_1,n_2),(n_2,s),W_{n_2}(n_1)x + T_{n_2}(n_1)]$
   - create $req[(n_1,n_2),(n_2,n_2),W_{n_2}(n_1)x + T_{n_2}(n_1)]$ if $r_{n_2}(n_1) = 1$
   - wait acknowledgements

3 An example

Let ϕ be the FPNA whose neuron resources, communication links and configured local connections are shown in figure 1, and for which:

- $i_{n_3} = i_{n_4} = i_{n_5} = ((x,x') \mapsto x + x')$
- $f_{n_3} = f_{n_4} = (x \mapsto \tanh(x))$ and $f_{n_5} = (x \mapsto x)$
- $\theta_3 = 2.1$, $\theta_4 = -1.9$, $\theta_5 = 0$, $a_3 = 4$, $a_4 = 2$, $a_5 = 2$, $c_1 = 2$, $c_2 = 1$
- $\forall\,(n_i,n_j) \in \mathcal{E}$ $W_{(n_i,n_j)} = i * j$ $T_{(n_i,n_j)} = i + j$

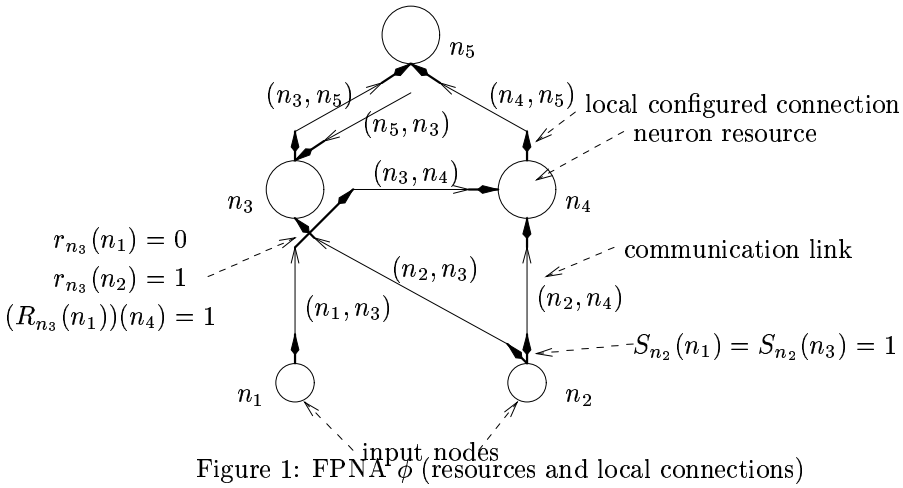

Figure 1: FPNA ϕ (resources and local connections)

Global FPNA inputs are chosen: $x_{n_1}^{(1)} = 1.5$, $x_{n_1}^{(2)} = -0.8$, $x_{n_2}^{(1)} = 1.1$. Moreover $c_{n_3} = c_{n_4} = c_{n_5} = 0$, $x_{n_3} = \theta_3 = 2.1$, $x_{n_4} = -1.9$, $x_{n_5} = 0$. The asynchronous sequential computation method applies to ϕ as follows:

Initial tasks: $[(n_1, n_3), 1.5]$, $[(n_1, n_3), -0.8]$, $[(n_2, n_3), 1.1]$ and $[(n_2, n_4), 1.1]$.

Sequential processing:

1. Task $[(n_1, n_3), 1.5]$ is first processed: $x' = 1.5\, W_{n_3}(n_1) + T_{n_3}(n_1) = 8.5$, task $[(n_3, n_4), 8.5]$ is created since $(R_{n_3}(n_1))(n_4) = 1$, and the value 8.5 is not received by the neuron in n_3 since $r_{n_3}(n_1) = 0$.
2. Task $[(n_1, n_3), -0.8]$ is similarly processed: task $[(n_3, n_4), 1.6]$ is created.
3. Task $[(n_2, n_3), 1.1]$ is processed: $x' = 11.6$ is received by the neuron in n_3, $c_{n_3} = 1$ (therefore $c_{n_3} < a_{n_3}$), $x_{n_3} = 13.7$.
4. Task $[(n_2, n_4), 1.1]$: neuron in n_4 receives $x' = 14.8$, $c_{n_4} = 1$, $x_{n_4} = 12.9$.
5. Task $[(n_3, n_4), 8.5]$: neuron in n_4 receives $x' = 109$, $c_{n_4} = 2$, $x_{n_4} = 121.9$. Now $c_{n_4} = a_{n_4}$: reset ($c_{n_4} = 0$, $x_{n_4} = -1.9$), $y = \tanh(109) \simeq 1$, task $[(n_4, n_5), 1]$ is created.
6. Task $[(n_3, n_4), 1.6]$: $x' = 26.2$, $c_{n_4} = 1$, $x_{n_4} = 24.3$.
7. Task $[(n_4, n_5), 1]$: $x' = 29$, $c_{n_5} = 1$, $x_{n_5} = 29$.

The asynchronous parallel computation method may apply to ϕ in *different ways* that depend on the scheduling policy of each neural resource. A possible parallel computation might be the following:

Initialization: The initial set of requests is

$$\{req[(n_1, n_1), (n_1, n_3), 1.5], req[(n_1, n_1), (n_1, n_3), -0.8],$$

$$req[(n_2, n_2), (n_2, n_3), 1.1], req[(n_2, n_2), (n_2, n_4), 1.1]\}$$

Parallel processing progress:

1. Resources (n_1, n_3), (n_2, n_3) and (n_2, n_4) may work concurrently, so that three requests are processed in parallel:

- Request $req[(n_1,n_1),(n_1,n_3),-0.8]$ is processed (chosen among two requests towards (n_1,n_3)): an acknowledgement is sent to (n_1,n_1), request $req[(n_1,n_3),(n_3,n_4),1.6]$ is created since $-0.8\,W_{n_3}(n_1)+T_{n_3}(n_1)=1.6$ and $(R_{n_3}(n_1))(n_4)=1$
- Request $req[(n_2,n_2),(n_2,n_3),1.1]$: an acknowledgement is sent to (n_2,n_2), request $req[(n_2,n_3),(n_3,n_3),11.6]$ is created since $1.1\,W_{n_3}(n_2)+T_{n_3}(n_2)=11.6$ and $r_{n_3}(n_2)=1$.
- Request $req[(n_2,n_2),(n_2,n_4),1.1]$: acknowledgement sent to (n_2,n_2), request $req[(n_2,n_4),(n_4,n_4),14.8]$ created.

2. Resources (n_1,n_3), (n_3,n_4), (n_3,n_3) and (n_4,n_4) work concurrently:
   - Since $req[(n_1,n_3),(n_3,n_4),1.6]$ has been acknowledged, resource (n_1,n_3) may process $req[(n_1,n_1),(n_1,n_3),1.5]$: acknowledgement sent to (n_1,n_1), request $req[(n_1,n_3),(n_3,n_4),8.5]$ created.
   - Request $req[(n_1,n_3),(n_3,n_4),1.6]$: acknowledgement sent to (n_1,n_3), request $req[(n_3,n_4),(n_4,n_4),26.2]$ created.
   - Request $req[(n_2,n_3),(n_3,n_3),11.6]$: acknowledgement sent to (n_2,n_3), $c_{n_3}=1$ ($c_{n_3}<a_{n_3}$), $x_{n_3}=13.7$, no created request.
   - Request $req[(n_2,n_4),(n_4,n_4),14.8]$: acknowledgement sent to (n_2,n_4), $c_{n_4}=1$ (therefore $c_{n_4}<a_{n_4}$), $x_{n_4}=12.9$.
3. Resource (n_3,n_4) has a request to process, but it must first wait for the acknowledgement of:
   - Request $req[(n_3,n_4),(n_4,n_4),26.2]$: an acknowledgement is sent to (n_3,n_4), $c_{n_4}=2$ (therefore $c_{n_4}=a_{n_4}$), $x_{n_4}=39.1$, request $req[(n_4,n_4),(n_4,n_5),\tanh(39.1)]$ is created, reset ($c_{n_4}=0$, $x_{n_4}=-1.9$).
4. ...

This simple example is not useful: one would expect ϕ to map the global input values to some global output values (for example computed by n_5). Yet it shows several specific characteristics of FPNA computing.

4 Correctness

The above computation algorithms appear as quite different, and they are based on local processings. The first question is whether they allow to define a global behaviour for the whole FPNA. This correctness problem may be expressed as follows: do the asynchronous sequential and parallel computations halt ?

A first simple result answers this question when it is dealt with the asynchronous sequential computation:

Theorem 1 *If ϕ is a feedforward FPNA, then its asynchronous sequential computation halts.*

The proof is straightforward, since the task generation process is locally finite, and follows the oriented edges of $\mathcal{G}_D(\phi)$. See [Gir99b] for more details.

4.1 Deadlock in the asynchronous parallel computation

The case of the asynchronous parallel computation is a bit more tricky. It is linked with, but not equivalent to, the notion of deadlock. Though this notion is rather standard when it is dealt with distributed communication protocols, it must be first defined for the above parallel algorithm.

Definition 3 *A FPNA is said to be in deadlock conditions during its asynchronous parallel computation if there exist resources $\varrho_0, \ldots, \varrho_p$ such that each ϱ_i is processing a request, and if this processing may not halt unless ϱ_{i+1} terminates the request processing it is performing (considering that $\varrho_{p+1} = \varrho_0$).*

Then a rather simple result ensures that the asynchronous parallel computation applies to feedforward FPNAs without deadlock.

Theorem 2 *If ϕ is a feedforward FPNA, then it may not be in deadlock conditions during its asynchronous parallel computation.*

The proof is easy. It is based on an equivalence between deadlock conditions and the existence of a cycle in a subgraph of $\mathcal{G}_D(\phi)$ (graph inferred by simultaneously processed requests). See [Gir99b] for more details.

4.2 Conditions of parallel non-termination

The main result is the link between deadlock conditions and non-termination. It ensures that deadlock is the only possible cause of non-termination.

Theorem 3 *If ϕ is a FPNA whose asynchronous sequential computation halts, then its asynchronous parallel computation does not halt if and only if it is in deadlock conditions.*
sufficient condition*: obvious (see definition 3)*
necessary condition *(proof outline, see [Gir99b] for full details):*

1. *The set of requests may be partially ordered.*
2. *The number of requests is finite (there exists an injection towards the set of sequential tasks).*
3. *There is a request whose processing does not halt.*
4. *A deadlock cycle may be built in the set of the direct and indirect successors of a non-terminating request.*

A straightforward consequence is that both asynchronous sequential and parallel computations of a feedforward FPNA halt. Other computation algorithms are defined in [Gir99b] for recurrent FPNAs. Similar correctness results are available (theorem 2 may be extended to recurrent FPNAs under some circuitous conditions). Moreover the study of [Gir99b] shows how simple conditions ensure that all defined computation methods are equivalent and deterministic.

5 Conclusion

The FPNA framework is a neural computation paradigm that has been defined in order to fit direct digital hardware implementations. It has given rise to a complete theoretical model for topologically constrained neural computation. This report presents the basis of this model: FPNAs are defined, two computation methods are described, and their correctness is asserted. FPNAs have been used within numerous neural applications: they allow to perform the computations of standard neural networks with the help of topologically simplified architectures, so that efficient hardware implementations are easily obtained ([Gir99b, Gir00, Gir99d]). Current theoretical studies analyse the FPNA concept in terms of weakly parameterized models for machine learning.

References

[Gir99a] B. Girau. Dependencies of composite connections in Field Programmable Neural Arrays. Research report NC-TR-99-047, NeuroCOLT, Royal Holloway, University of London, 1999.

[Gir99b] B. Girau. *Du parallélisme des modèles connexionnistes à leur implantation parallèle.* PhD thesis n° 99ENSL0116, ENS Lyon, 1999.

[Gir99c] B. Girau. FPNA, FPNN: from programmable fields to topologically simplified neural networks. Research report 99.R.019, LORIA INRIA-Lorraine, 1999.

[Gir99d] B. Girau. Synchronous FPNNs: neural models that fit reconfigurable hardware. Research report 99.R.143, LORIA INRIA-Lorraine, 1999.

[Gir00] B. Girau. Neural networks on FPGAs: a survey. In *Proc. Neural Computation*, 2000. To be published.

[Mor95] N. Morgan. Programmable neurocomputing systems. In *The Handbook of Brain Theory and Neural Networks*, pages 764–768. MIT Press, 1995.

[NS92] T. Nordström and B. Svensson. Using and designing massively parallel computers for artificial neural networks. *Journal of Parallel and Distributed Computing*, 14(3):260–285, 1992.

[Sal94] V. Salapura. Neural networks using bit-stream arithmetic: a space efficient implementation. In *Proc. IEEE Int. Conf. on Circuits and Systems*, 1994.

[vDJST93] M. van Daalen, P. Jeavons, and J. Shawe-Taylor. A stochastic neural architecture that exploits dynamically reconfigurable FPGAs. In *Proc. of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 202–211, 1993.