# FIST: A Framework to Interleave Spiking neural networks on CGRAs

Tuan Ngyen [†], Syed M. A. H. Jafri [*†‡], Masoud Daneshtalab[†‡], Ahmed Hemani[‡]
, Sergei Dytckov[‡], Juha Plosila[†‡], and Hannu Tenhunen[†‡]
[*]Turku Centre for Computer Science
[†]University of Turku, Finland
[‡]Royal Institute of Technology, Sweden

***Abstract*—Coarse Grained Reconfigurable Architectures (CGRAs) are emerging as enabling platforms to meet the high performance demanded by modern embedded applications. In many application domains (e.g. robotics and cognitive embedded systems), the CGRAs are required to simultaneously host processing (e.g. Audio/video acquisition) and estimation (e.g. audio/video/image recognition) tasks. Recent works have revealed that the efficiency and scalability of the estimation algorithms can be significantly improved by using neural networks. However, existing CGRAs commonly employ homogeneous processing resources for both the tasks. To realize the best of both the worlds (conventional processing and neural networks), we present FIST. FIST allows the processing elements and the network to dynamically morph into either conventional CGRA or a neural network, depending on the hosted application. We have chosen the DRRA as a vehicle to study the feasibility and overheads of our approach. Synthesis results reveal that the proposed enhancements incur negligible overheads (4.4% area and 9.1% power) compared to the original DRRA cell.**

## I. Introduction and motivation

Recently, the increasing speed and performance requirements of embedded applications, coupled with the demands for flexibility and low non-recurring engineering costs, have made reconfigurable hardware a very popular implementation platform. The reconfigurable architectures can be classified on the basis of granularity i.e. number of bits that can be explicitly manipulated. Coarse Grained Reconfigurable Architectures (CGRAs), provide operator level configurable functional blocks, word level datapaths, and very area-efficient routing switches. Therefore, compared to the fine-grained architectures (like FPGAs), CGRAs require lesser configuration memory and configuration time (two or more orders of magnitude [1]). As a result, CGRAs achieve a significant reduction in area (from 66 % to 99.06 % [2]) and energy consumed per computation (from 88 % to 98 % [2]), at the cost of a loss in flexibility compared to bit-level operations. Therefore, CGRAs have been a subject of intensive research since the last decade [2], [3].

Today, CGRAs host multiple applications simultaneously on a single platform. Each application can potentially have different requirements (e.g. MPEG4 decoder requires exact calculation while edge detection can tolerate approximations). For the estimation problems, *neural network* promise higher efficiency and scalability compared to conventional processing algorithms [4]. However, the existing CGRAs lack the support to simultaneously provide conventional and the neural networks based processing [4]. As a solution to this problem, we present FIST. The proposed architecture allows to efficiently interleave the conventional processing with estimation (using neural networks). For the applications that require exact calculations, the device behaves like a normal CGRA, with MACs/ALUs connected via circuit switched interconnect. When an application, that can tolerate approximate results, enters the platform the device dynamically morphs into a neural network.

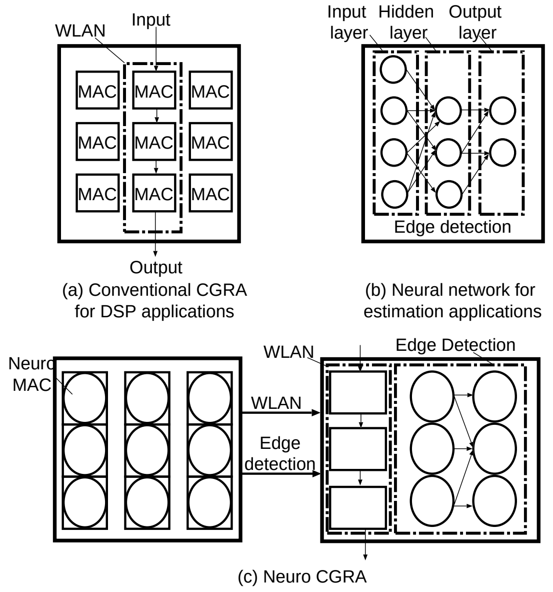

Fig. 1. Motivation for FIST

To visualize our technique, consider Fig. 1 (a) that shows simple WLAN transmitter mapped to a conventional CGRA [5], [6]. The figure shows a typical scenario, where multipliers/accumulators are connected in a pipelined fashion to produce outputs. Fig. 1 (b) depicts edge detection realization on a neural network. A neural network typically consists of three elements: (i) neurons (the processing elements), (ii) synapse (the interconnect network), and (iii) weights. Usually a neural network consists of three layers of neurons (input layer, hidden layer, and output layer). Each neuron layer performs calculation on the weights based on the neuron model. The results calculated in one layer are communicated to the neurons in the next layer (to the right). Based on the joint calculation of the neurons, the edges in an image are detected.

Fig. 1 (c) shows functionality of the proposed CGRA. Each processing element can act as either as conventional MAC or a neuron. When WLAN and edge detection request platform resources, the platform dynamically creates a different partition for each application. For WLAN and edge detection the platform resources morph into conventional and neural network, respectively.

This paper is organized as follows. In Section II, a brief survey of various platforms, that use neural networks, is presented. In Section III, an overview of the CGRA platform, used in this paper is described. In Section IV, details of the proposed method are presented. In Section V, we evaluate the benefits and redundancies imposed by our method on an actual CGRA. Finally, in Section VI, we summarize our contributions and suggest directions for future research.

## II. Related work and contributions

Hardware implementations of neural networks has been a subject of intensive research since the last two decades [7]. In this section we will review only the most prominent work dealing with neural network implementation on reconfigurable architectures.

Much of the work implementing neural networks on reconfigurable architectures deals with FPGAs. In particular it focuses on how various algorithms can be realized using neural networks. Krips et al. [8] presented an FPGA based implementation of neural networks to track video images. Yang and Paindavoine [9] proposed a technique to track faces using neural networks. Maeda and Tada [10] developed a neural network model to support online learning. Himavathi et al. [11] used layer multiplexing technique to implement multilayer feed-forward networks on FPGA.

Recent works have revealed that for estimation or approximate problems, the neural networks in particular offer higher energy efficiency and speedup compared to conventional algorithms. They have shown that neural networks can also be used as accelerators. Chen et al. [12] provide a detailed discussion of hardware and software based accelerators (using neural networks) with special emphasis on memory efficiency. Esmaeilzadeh et al. [13] presented an ASIC based accelerator to efficiently speed up approximate programs. Chakradhar et al. [14] proposed an FPGA based configurable coprocessor that dynamically configures the hardware to provide the best achievable throughput (on the available resources). The main idea of this paper, i.e to interleave conventional processing with neural networks on a unique platform, is inspired from this work. The major difference is that our work focuses on CGRAs, that have significantly different architectures and lesser flexibility, compared to FPGAs. Hassan et al. [15] and Asad et. al. [16] explored the feasibility of implementing spiking neural networks on a CGRA. However, they did not test the effectiveness of their technique for an actual actual algorithm (e.g. edge detection). Moreover, they were geared for speed while the focus of this paper is the fault tolerance achievable by neural networks. Compared to these works, this paper will Zhengrong [17] showed that the spiking neural networks (when compared to Sobel and Prewitt filter) are more efficient in detecting edges (in terms of accuracy), can be easily customized to detect specific parts of an image, and can tolerate high levels of noise. Zhengrong's work has motivated us to implement edge detection using spiking neural networks.

The related work reveals that most of the work on neural networks deals with their realization on FPGAs. The works that do use neural networks to enhance efficiency, typically employ FPGAs or ASICs. None of the existing works evaluates the feasibility of implementing neural networks on CGRAs.

**Compared to the related work, this paper has three major contributions:**

1) We present FIST that allows to interleave the conventional calculations with neural networks;
2) We propose a neural network translator that provides a framework to map neural network on CGRAs; and
3) We evaluate benefits and overheads of implementing the proposed technique on an actual CGRA.

## III. System Overview

We have chosen the Dynamically Reconfigurable Resource Array (DRRA) [18], [19], [20] as a vehicle to evaluate feasibility of implementing neural networks on an actual CGRA. Nevertheless, it seems that the results should be applicable to most grid based CGRAs as well. The motivation for choosing DRRA is that it is well documented, its bandwidth has been tested on demanding industrial applications (with Huawei) [21], and we have available its architectural details from RTL codes to physical design.

### *A. DRRA configuration flow*

As shown in Fig. 2, DRRA is programmed in two phases (off-line and on-line) [22], [23], [24]. The configware (binary) for commonly used DSP functions (FFT, FIR filter etc.) is written in VESYLA (HLS tool for DRRA) and stored in an off-line library. For each function, multiple versions, with different degree of parallelism, are stored. The library, thus created, is profiled with frequencies and worst case time of each version. To map an application, its (simulink type) representation is fed to the compiler. The compiler, based on the available functions (present in library) constructs the binary for the complete application (e.g. WLAN). Since the actual execution times are unknown at compile-time, the compiler sends all the versions (of each function), meeting deadlines, to the run-time configware memory. To reduce memory requirements for storing multiple versions, the compiler generates a compact representation of these versions. Details of compression algorithm and how it is unraveled are given in [22]. The compact representation is unraveled (parallelized/serialized) dynamically by the run-time resource manager (running on LEON3 processor).

### *B. DRRA Computation Layer*

DRRA computational layer is shown in Fig. 3. It is composed of four elements: (i) Register Files (reg-files), (ii) morphable Data Path Units (DPUs), (iii) circuit-Switched Boxes (SBs), and (iv) sequencers. The reg-files store data for DPUs. The DPUs are functional units responsible for performing computations. SBs provide interconnectivity between different

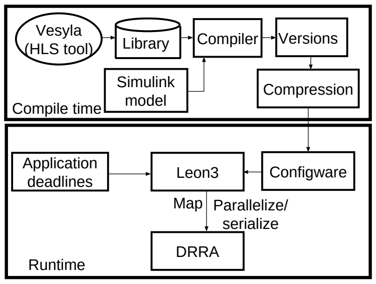

Fig. 2. Configuration Model

components of DRRA. The sequencers hold the configware which corresponds to the configuration of the reg-files, DPUs, and SBs. Each sequencer can store up to 64 36-bit instructions and can reconfigure the elements only in its own cell. As shown in Fig. 3, a *cell* consists of a reg-file, a DPU, SBs, and a sequencer, all having the same row and column number as a given cell. The configware loaded in the sequencers contains a sequence of instructions (reg-file, DPU, and SB instructions) that implements the DRRA program.

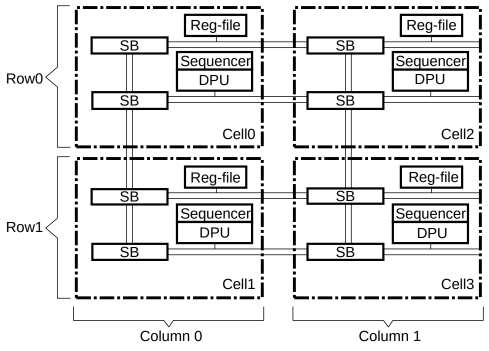

Fig. 3. DRRA computation layer

### C. DRRA Storage Layer (DiMArch)

DiMArch is a distributed scratch pad (data/configware) memory that complements DRRA with a scalable memory architecture. Its distributed nature allows a high speed data and configware access to the DRRA computational layer (compared to the global configuration memory). Further discussion of DiMArch is beyond the scope of this paper and for details we refer to [19].

## IV. Neural networks/DRRA integration

In this section, we will present the neural network model chosen for this paper followed by its implementation on DRRA.

### A. Neural network model

For this work we have chosen Spiking Neural Networks (SNN). The motivation for choosing SNNs is that they are the latest generation of neural networks models that overcome the computational power of other neural networks generations [25].

For further details about the SNNs an interested reader can refer to [26]. For understanding, Fig. 4 depicts a simple neuron model. The neurons are connected in one-to-many fashion. In the figure, the neuron 4 receives input spikes from neurons 1, 2, and 3. Each connection is characterized by a weight (Wt1, Wt2, and Wt3). When a spike is generated, depending on the neuron model and weight of the connection, the neuron performs calculations.

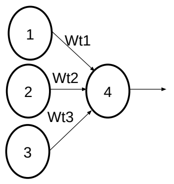

Fig. 4. A simplified neuron model

We have chosen a simple and the widely used model called leaky Integrate and Fire (I & F), to model a neuron. The model is given by the Equation 1:

$$dv/dt = (Wt * I) + a - (b * V). \tag{1}$$

Where the potential $V$ integrates input spikes $I$ and leaks over time with $-bV$ component. $Wt$ is the weight of the connection coefficient, $a$ determines equilibrium point, and coefficient $b$ is the speed of leakage. When the threshold potential is reached, a neuron outputs a spike and its potential is reset.

### B. Neural network realization on DRRA

To realize neural networks on DRRA, we have embedded a dedicated hardware, called neuroDPU, with each DPU of DRRA. As a result of our enhancements, the DPU can be configured to either normal or neuron mode. The details of the normal mode can be found in [2], here we only concentrate on the neuron mode. Fig. 5 shows how the DPU functions in neuron mode. The figure in particular illustrates how simple neural network, shown in Fig. 4, is implemented on DRRA. To mimic parallelism in neural networks, we have employed time multiplexing. Where a time slot is reserved for each transmitting neuron (connected to the receiving neuron). In Fig. 5 (a), the transmitting DRRA cell implements three neurons. A register from the reg-file is reserved for each neuron. In every cycle, the neuroDPU receives an input from one of the three registers (representing the neurons) in round robin fashion. The neuroDPU applies the Equation 1 to the input and stores the result in the dedicated register for accumulation. If the result is greater than pre-defined threshold, 1 is stored in the spike register (otherwise 0 is stored). Since the transmitting cell implements three neurons, the value of the spike register is sent to the receiving cell after every three cycles. The receiving cell stores the value of the spike register in a reserved location (at first address), of the reg-file. Fig. 5 (b) shows processing done by the receiving neuron. In each cycle, the neuroDPU of the receiving cell extracts a bit from spike register (stored in its reg-file) and the weight of the corresponding connection. It applies the $I\&F$ model to the inputs and generates the outputs.

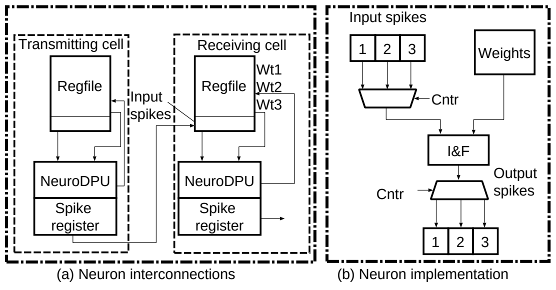

Fig. 5. Neuron realization on DRRA

## C. Clustering for scalability

In Section IV-B we showed how a simple neural network (containing three neurons) can be realized on DRRA. In this section, we will explain how a large scale neural network can be implemented in a scalable manner considering the architectural details of DRRA. Neural networks may require one to many communication between large number of neurons. To implement these connections on DRRA, we had to consider two architectural properties: (i) every DRRA component has only two read/write ports and (ii) a DRRA component can be directly connected to a component, at most three hops away. Since these architectural characteristics were designed after a careful evaluation of area/power trade-off, we decided not to modify them. The one-to-many connectivity was realized by using time division multiplexing. In the proposed approach a specific time slot is assigned to each pair of neurons. To allow a scalable solution, we have chosen a hierarchical clustered approach shown in Fig. 6.

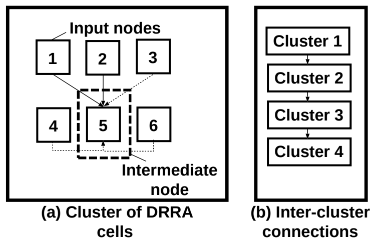

Fig. 6. Inter neural communication realization

Fig. 6 (a) shows a cluster of 6 DRRA cells. In the cluster one of the cells is chosen as an intermediate node. The intermediate node receives data from the 6 cells in the cluster (including itself). To allow connections with 6 cells on a 2-port component, we exploit time division multiplexing combined with partial and dynamic reconfiguration. In the overall process the intermediate cell receives data from 2 cells at a time and then shifts to the other cells. The process continues till the data from all the cells is received. Once every cluster has received inputs, the intermediate nodes communicate with each other serially to ensure one to many connectivity. Fig. 7 shows the instructions in DRRA sequencers (needed to implement the time division multiplexing). The figure shows that 5 cycles are needed to collect information from all the DRRA cells in the cluster. It should be noted that 2 additional cycles are required to reconfigure the circuit switched network. The inter cluster communication takes two cycles for reconfiguration and an additional cycle to transmit data. Each cell in the cluster can represent arbitrary neurons, depending on the application requirements.

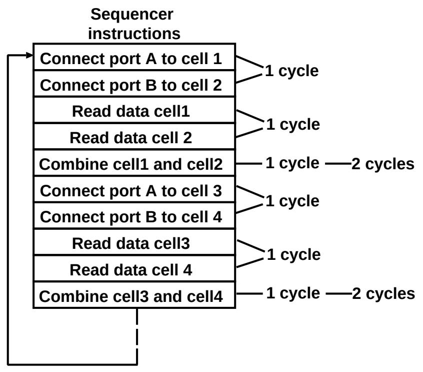

Fig. 7. Sequencer instructions to implement time division multiplexing

## D. Architectural integration

To realize FIST we have modified the configuration flow shown in the Fig. 2. We have added another block to generate the configware for the estimation algorithms (shown by the dotted box). The key component of the new block is a translator. The translator takes three inputs: (i) application, (ii) weights, and (iii) application to generate the reg-file, SB, and DPU instructions for DRRA (see Section III-B).

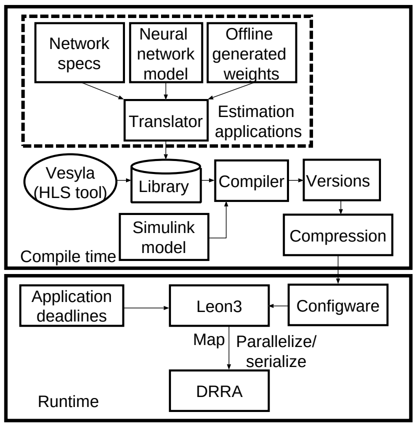

Fig. 8. Modified programming flow

Fig. 9 depicts how each input (to the translator) is represented, considering a neural network composed of 8 neurons. The neural network application is represented by a *directed acyclic graph*, similar to [27]. Where each node represents a neuron and the edges represent the interconnection between the two nodes. The motivation for choosing the directed acyclic graph is that it easily represents most of the neural network applications and can be easily converted to linked list for automated application generation. The weights are stored as a look-up table. For each neuron, the weights of all connected neurons are stored. If there is no connection between a pair of neurons (e.g. neuron 4 and 5 in the figure), 0 is stored. Finally, the network specifications in form the cluster size and the number of neurons in each cluster are provided to the translator. In the simplest case i.e. $clustersize = 1$, only one DRRA cell for each neuron layer (input, hidden and output layer) is sufficient. If $clustersize > 1$, the required DRRA cells, $DRRA_c$, are given by $DRRA_c = clusters_p - clusters_i$. Where $clusters_p$ and $clusters_i$ denote respectively the processing and the intermediate clusters.

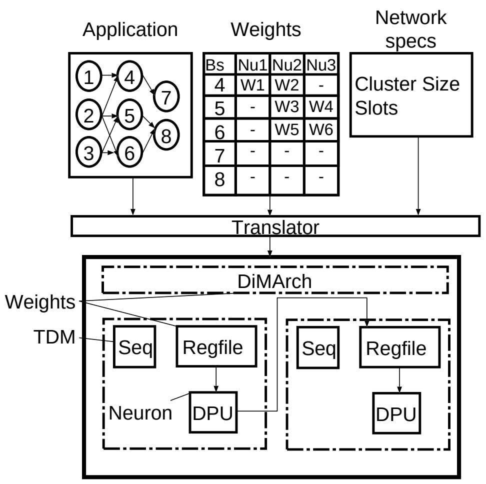

Fig. 9. Translator functionality overview

The translator functionality is shown in Algorithm 1. The translator works in three steps: (i) DPU instruction generation, (ii) reg-file instruction generation, and (iii) SB instruction generation. The algorithm shows the translator functionality when $clustersize = 1$. If an application requires more clusters, the same algorithm can be applied to generate instructions for each cluster. In the first step, a separate DPU instructions is generated. In the second step, the reg-file for each cell is initialized. One of the register in each reg-file is reserved to store spikes. Since each register of the reg-file is 16-bit wide, the spike register can contain up to 16 spikes. The weights for each neuron connection is loaded to the register files. If a neuron is connected to greater than 63 neurons, the weights are stored in DiMArch memory. The details of the data transfer from DiMArch to reg-files are beyond the scope of this paper and for details an interested reader can refer to [19]. The total number of neurons per reg-file is dependent on the cluster size. In the third step, the switch-box instructions are generated. If the $clustersize = 1$ only three interconnect instructions (one for each layer) are needed. Otherwise, additional interconnect instructions to interconnect the clusters are also generated (similar to Fig. 7).

To mimic the neuron functionality, we have enhanced the functionality of the Data Path Unit (DPU). With each DPU we have added a special unit to perform I & F model. The weights are stored in reg-files and the DiMArch. The reg-file are also used to store spikes from other interconnected neurons. Since the spiked neural networks require timing information, we have added a global counter. To realize the complex neural network the sequencers are programmed to implement TDM.

```
Input: application, weights, network specifications ;
Output: DPU, reg-file, and SB instructions ;
/* Generate DPU instructions */
if Neural application exists then
    DPU=neural mode ;
else
    DPU=normal mode ;
end
/* Generate Reg-file instructions */
for $j \leftarrow 0$ to $N$ do
    /* $N$ is the number neurons in application */
    Spike register delay= $con_j$ ;
    /* $con_j$ is the connections per DRRA cell */
    for $k \leftarrow 0$ to $con_j$ do
        i=62 ;
        $Reg - file_i$= $Wt_k$ ;
        i- -;
        if $i < 0$ then
            store Wt in DiMArch ;
            Delay=0
        end
        Reg-file = Delay ;
        Delay ++ ;
    end
end
/* Generate SB instructions */
if Cluster size $< 2$ then
    Generate SB instructions to connect three layers ;
else
    Generate SB instructions for clusters;
    Generate SB instruction for inter-cluster communications;
end
```

**Algorithm 1:** Translator functionality

## V. Results

### A. Edge detection on DRRA

To evaluate the efficacy of our approach, we implemented edge detection on DRRA, using SNNs.

*1) Edge detection using Spiking Neural Networks:* To evaluate the efficacy of implementing neural networks, we have chosen the neural network based edge detection algorithm presented in [28]. The motivation for choosing spiking neural network based edge detection is that it is more flexible (e.g. can be easily customized to detect white lines in an image) resilient to noise compared to Sobel and Prewitt filters [17]. The overall technique is shown in Fig. 10. The techniques relies on three layers of neurons. The first layer consists of input neurons. The input neurons receive the image and apply the I & F processing to the received pixel values. The result is sent to the middle or intermediate layer of neurons. The intermediate layer has four parallel arrays of neurons. It should be noted that for clarity, the figure only shows one neuron in each array. Each of these layers perform the processing for up, down, left and right edges respectively and is connected to the input layer by differing weights. The processed information is transmitted to the neuron in the output layer. This neuron integrates the outputs from the four neurons of the middle layer to detect an edge. For more details about the theory behind neural network based edge detection, an interested reader can refer to [28], [17].

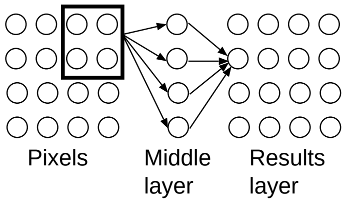

Fig. 10. Edge detection using neural networks

*2) Implementation of Edge detection on DRRA:* Fig. 11 illustrates how edge detection is mapped to DRRA. For clarity, the figure only shows the mapping for an image with only 2*2 pixels. Four neurons in receptor layer are statically stored in registers 1, 2, 3, and 4 of the cell 1. The pixel values are sequentially transferred to the DPU of cell 1. The DPU applies I & F model to the received pixel value and generates two outputs: (i) spike and (ii) accumulation value. The technique to generate these spikes was already shown in Section IV. The 4 neurons of the middle layer are mapped to the cell 2 of DRRA. In each cycle, the weight of the connection and the spike from the spike register is sent to the DPU. The DPU behaves the same way, as the DPU of cell 1, and sends the spike register value to the output layer. The output layer consists of a single neuron connected by the four neurons of the middle layer. Therefore, four registers are reserved for the four connections. The DPU of the output layer performs the I & F model and generates the final output.

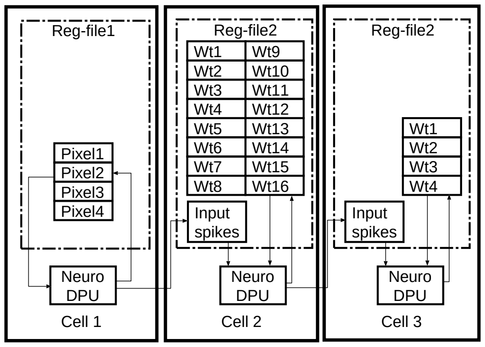

Fig. 11. Edge detection on DRRA

## B. Spiking neural network benifits

The main motivation to implement the SNN on the DRRA was their superiority to handle estimation problems. To test the feasibility of our approach, we applied edge detection on Lena's image, using three methods: (i) Sobel filter, (ii) Prewitt filter, and (iii) SNN (implemented on DRRA). For evaluating the tolerance to noise, we applied Poisson and salt and pepper noise to the images. The results are shown in Fig. 12. The figure reveals that on the original images, all the algorithms detect edges correctly. After applying the noise, it can be seen that both the Sobel and Prewitt filters detect false edges. In particular, they behave poorly on the salt and pepper noise. SNNs inherently filter the noise and successfully detect the edges.

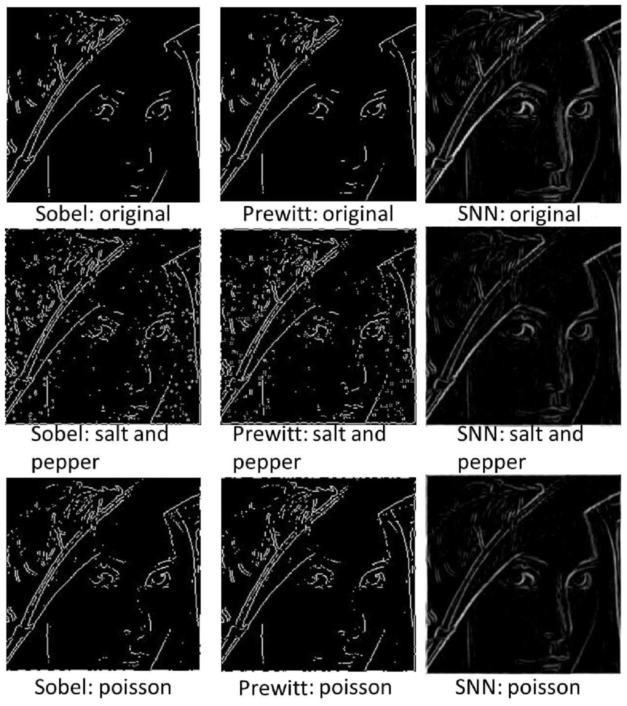

Fig. 12. Edge detection using various edge detection operators

## C. Timing vs accuracy tradeoff

To detect the effect of accuracy on timing overheads (of the SNNs), we performed edge detection with different *iterations*, using images of various sizes. Where an iteration is defined as the number of times the Equation 1 integrates. The accuracy of edge detection depends on the number of iterations. Table I and Fig. 13, show the time needed to detect edges with a single cluster, containing 3 cells. Where NN1, NN40, NN80, NN150, and NN200 denote respectively 1, 40, 80, 150, and 200 iterations. For the tested images, 200 iterations was found to be the upper bound, since no further accuracy was possible. The key thing to notice is that even with 3 DPUs the neural network based edge detection algorithm (with 200 iterations) can determine the edges of an image, containing up to in $10K$ pixels, in 32000000 cycles. Since DRRA operates at 500 MHz, the edges of the image can be calculated in 0.8 secs. Of course with more DPUs we can process a higher quality image in real time. As a reference, we have also given the time to calculate the edges using Sobel filter. It can be seen that the higher accuracy and noise tolerance comes at a cost of greater timing overhead. In the future work, we plan to decrease this overhead by using multiple clusters to exploit the inherent parallelism offered by the SNNs.

TABLE I
CYCLE REQUIRED FOR DIFFERENT ITERATIONS OF SPIKING NEURAL NETWORKS

| Images | Sobel cycles | NN40 cycles | NN80 cycles | NN150 cycles | NN250 cycles |
|---|---|---|---|---|---|
| 5*5 | 747 | 16000 | 32000 | 6000 | 80000 |
| 10*10 | 5312 | 64000 | 128000 | 24000 | 320000 |
| 15*15 | 14027 | 144000 | 288000 | 54000 | 720000 |
| 20*20 | 26892 | 256000 | 512000 | 96000 | 1280000 |
| 50*50 | 191232 | 1600000 | 3200000 | 600000 | 8000000 |
| 100*100 | 797132 | 6400000 | 12800000 | 2400000 | 32000000 |
| 150*150 | 1818032 | 14400000 | 28800000 | 5400000 | 72000000 |
| 200*200 | 3253932 | 25600000 | 51200000 | 9600000 | 128000000 |
| 400*400 | 13147532 | 102400000 | 204800000 | 38400000 | 512000000 |
| 1000*1000 | 82668332 | 640000000 | 128000000 | 240000000 | 3200000000 |

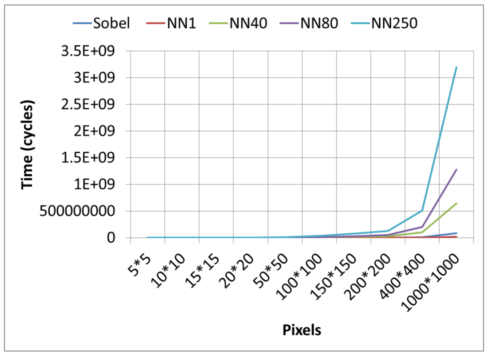

Fig. 13. Cycles required for different iterations of Spiking Neural Networks

## D. Overhead analysis

To check the overhead imposed by the additional I & F unit, we synthesized it using 65 $nm$ technology with frequency of 500 MHz. Table II shows the obtained results. It can be seen that the proposed enhancement incurs 9.1% area and 4.2% power overheads. Further evaluation revealed that the most of overhead (95% area) results from the fixed-point multipliers used to implement Equation 1. This overhead can be significantly reduced either by using a look up table or reusing the existing multiplier in the DPU (which will be considered in future).

TABLE II
AREA AND POWER CONSUMPTION OF I & F UNIT

| | I & F | DRRA cell | Overhead (%) |
|---|---|---|---|
| Power $mW$ | 6.44 | 70.40 | 9.1 |
| Area $\mu m^2$ | 50920 | 1199506 | 4.2 |

## VI. CONCLUSION

In this paper, we have presented FIST, to efficiently host estimation algorithms (using neural networks) along side normal calculations algorithms on a CGRA. The overall architecture relies on dedicated hardware blocks (to mimic the neuron) and a modified control flow (to translate the neural network application model to DRRA bitstream). To implement the neural networks, we embedded additional hardware to mimic the neuron functionality. The synthesis/simulation results using edge detection reveal that proposed architecture incurs acceptable overheads (4.4% area and 9.1% power). Future research on FIST will involve development of other algorithms. Additionally, we also plan to test the feasibility of using neural networks, with online learning, as accelerators (for large holographic images).

## REFERENCES

[1] D. Alnajjar, H. Konoura, Y. Ko, Y. Mitsuyama, M. Hashimoto, and T. Onoye, “Implementing flexible reliability in a coarse-grained reconfigurable architecture,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,*, vol. PP, no. 99, pp. 1–1, 2012.

[2] M. A. Shami, “Dynamically reconfigurable resource array,” Ph.D. dissertation, Royal Institute of Technology (KTH), Stockholm, Sweden, 2012. [Online]. Available: web.it.kth.se/~hemani/Athesis15.pdf

[3] Z. ul Abdin and B. Svensson, “Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing,” *Microprocess. Microsyst.*, vol. 33, no. 3, pp. 161–178, May 2009. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2008.10.003

[4] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, “Enerj: Approximate data types for safe and general low-power computation,” in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11. New York, NY, USA: ACM, 2011, pp. 164–174. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993518

[5] S. M. A. H. Jafri, O. Ozbak, A. Hemani, N. Farahini, K. Paul, J. Plosila, and H. Tenhunen, “Energy-aware CGRAs using dynamically reconfigurable isolation cells.” in *Proc. International symposium for quality and design (ISQED)*, 2013, pp. 104–111.

[6] S. Jafri, M. A. Tajammul, A. Hemani, K. Paul, J. Plosila, and H. Tenhunen, “Energy-aware-task-parallelism for efficient dynamic voltage, and frequency scaling, in cgras,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, 2013, pp. 104–112.

[7] J. Misra and I. Saha, “Artificial neural networks in hardware: A survey of two decades of progress,” *Neurocomput.*, vol. 74, no. 1-3, pp. 239–255, Dec. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.neucom.2010.03.021

[8] M. Krips, T. Lammert, and A. Kummert, “Fpga implementation of a neural network for a real-time hand tracking system,” in *Electronic Design, Test and Applications, 2002. Proceedings. The First IEEE International Workshop on*, 2002, pp. 313–317.

[9] F. Yang and M. Paindavoine, “Implementation of an rbf neural network on embedded systems: Real-time face tracking and identity verification,” *Trans. Neur. Netw.*, vol. 14, no. 5, pp. 1162–1175, Sep. 2003. [Online]. Available: http://dx.doi.org/10.1109/TNN.2003.816035

[10] Y. Maeda and T. Tada, “Fpga implementation of a pulse density neural network with learning ability using simultaneous perturbation,” *Neural Networks, IEEE Transactions on*, vol. 14, no. 3, pp. 688–695, May 2003.

[11] S. Himavathi, D. Anitha, and A. Muthuramalingam, “Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization,” *Neural Networks, IEEE Transactions on*, vol. 18, no. 3, pp. 880–888, May 2007.

[12] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14. New York, NY, USA: ACM, 2014, pp. 269–284. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541967

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, Dec 2012, pp. 449–460.

[14] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, “A dynamically configurable coprocessor for convolutional neural networks,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA ’10. New York, NY, USA: ACM, 2010, pp. 247–257. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815993

[15] H. Anwar, S. M. A. H. Jafri, S. Dytckov, M. Daneshtalab, M. Ebrahimi, and A. Hemani, “Exploring spiking neural network on coarse-grain reconfigurable architectures,” in *Proceedings of International Workshop on Manycore Embedded Systems*, ser. MES ’14. New York, NY, USA: ACM, 2014, pp. 64:64–64:67. [Online]. Available: http://doi.acm.org/10.1145/2613908.2613916

[16] S. Jafri, T. N. Gia, S. Dytckov, M. Daneshtalab, A. Hemani, J. Plosila, and H. Tenhunen, "Neurocgra: A cgra with support for neural networks," in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, July 2014, pp. 506–511.

[17] Z. Li, "Aerial image analysis using spiking neural networks with application to power line corridor monitoring," Ph.D. dissertation, Queensland University of Technology, 2011.

[18] M. A. Shami and A. Hemani, "Classification of massively parallel computer architectures," in *Proc. IEEE Int. Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, May 2012, pp. 344–351.

[19] M. A. Tajammul, S. M. A. H. Jafri, A. Hemani, J. Plosila, and H. Tenhunen, "Private configuration environments for efficient configuration in CGRAs," in *Proc. Application Specific Systems Architectures and Processors (ASAP)*, Washington, D.C., USA, 5–7 June 2013.

[20] S. Jafri, S. Piestrak, K. Paul, A. Hemani, J. Plosila, and H. Tenhunen, "Energy-aware fault-tolerant cgras addressing application with different reliability needs," in *Digital System Design (DSD), 2013 Euromicro Conference on*, Sept 2013, pp. 525–534.

[21] N. Farahini, S. Li, M. A.l Tajammul, M. A. Shami, G. Chen, A. Hemani, W. Ye, "39.9 GOPs/Watt multi-mode CGRA accelerator for a multi-standard base station," in *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, 2013.

[22] S. Jafri, A. Hemani, K. Paul, J. Plosila, and H. Tenhunen, "Compact generic intermediate representation (CGIR) to enable late binding in coarse grained reconfigurable architectures," in *Proc. International Conference on Field-Programmable Technology (FPT)*,, Dec. 2011, pp. 1 –6.

[23] S. M. A. H. Jafri, A. Tajammul, M. Daneshtalab, A. Hemani, K. Paul, P. Ellervee, J. Plosila, and H. Tenhunen, "Morphable compression architecture for efficient configuration in cgras," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, Aug 2014, pp. 42–49.

[24] S. M. Jafri, A. Tajammul, M. Daneshtalab, A. Hemani, K. Paul, P. Ellervee, J. Plosila, and H. Tenhunen, "Customizable compression architecture for efficient configuration in cgras," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, May 2014, pp. 31–31.

[25] H. Paugam-Moisy and S. M. Bohte, *Handbook of Natural Computing*. Springer-Verlag, Sep. 2009, ch. Computing with Spiking Neuron Networks. [Online]. Available: http://liris.cnrs.fr/publis/?id=4305

[26] E. Izhikevich, "Simple model of spiking neurons," *Neural Networks, IEEE Transactions on*, vol. 14, no. 6, pp. 1569–1572, 2003.

[27] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618

[28] Q. Wu, T. M. McGinnity, L. P. Maguire, A. Belatreche, and B. P. Glackin, "Edge detection based on spiking neural network model," in *ICIC (2)*, 2007, pp. 26–34.