ADS Report

Team 83

November 2023

Contents

1 Introduction

2 Architectures

2.1 Neuromorphic

2.1.1 Evolutionary Optimization for Neuromorphic Systems

An approach for utilizing evolutionary optimization to address this challenge called Evolutionary Optimization for Neuromorphic Systems.

2.1.2 DANNA2 - Dynamic AdaptiveNeural Network Arrays

Components

Neurons
The core compute unit in DANNA2 is the neuron. DANNA2 uses a leaky accumulate and fire approach with discrete time-steps and configurable linear charge leak. Each neuron accumulates charge from input spikes. At the end of a cycle, if the neuron exceeds its threshold, it fires, resets its charge to a resting state, and enters refractory. In the refractory period, the neuron may accumulate new charge, but it may not fire again until the refractory period is complete.

$$H_{kj}(t) = \sum_{i=1}^{N} w_i(t)x_i(t) + H_{kj}(t-1) - L_{kj}(t-1) \tag{1}$$

$$a_{kj} = f(H_{kj}(t)) \tag{2}$$

DANNA2 neurons are allowed integer thresholds in the range [0, 1023]. This represents 10 bits of unsigned integer precision. The refractory period for a neuron may be configured as up to 7 cycles after a fire. Optionally, the refractory may be set to 0 which means a neuron can fire on consecutive cycles with no effective refractory.

Synapses
In order to connect neurons together, DANNA2 uses explicit instances of synapses. A synapse is responsible for carrying fires from a pre-synaptic neuron to a post-synaptic neuron. Each synapse carries these fires with a configurable and possibly dynamic weight value. This weight value is the amount of charge which the post-synaptic neuron will accumulate. Additionally, a synapse may impose some configurable delay in transferring the fire.

In DANNA2, a synapse is allowed to have integer weights in the range $[256, 255]$ representing 9 bits of signed precision.

Spike Timing Dependent Plasticity
A common online learning algorithm for spiking neural networks is Spike Timing Dependent Plasticity (STDP). Many neuromorphic systems seek to include some form of online learning for in the field adaptation. DANNA2 offers a simplified generic STDP implementation using a look up table keyed on the time difference between a neuron and synapse fire.

$$\Delta t = t_{synapse\_fire} - t_{neuron\_fire} \tag{3}$$

STDP weight adjustments are assessed when either the neuron has just fired or the synapse has just fired. For example, when the neuron has just fired, the Δt is the time since the last synapse fire, and when the synapse has just fired, the Δt is the time since the last neuron fire.

$$W(i,j)_t = W(i,j)_{t-1} + S(\Delta t) \tag{4}$$

Arrays
In order to have a functional system, neurons must be connected together with synapses into directed graphs. These directed graphs are referred to as networks. With DANNA2, a network is implemented onto an array of DANNA2 elements.

Currently, there are two types of arrays – grid arrays and sparse arrays. With both array types, there are a few common concepts. Every element is given coordinates on a two dimensional plane, and typically each neuron is restricted to 24 incoming synaptic connections. Inputs to the array are representatively located along the first column of coordinates in the network, and outputs are in the last column.

Grid Arrays
DANNA2 grid arrays are similar to DANNA arrays. A grid array is a rectangular matrix of elements with restricted connectivity. In a grid array, one neuron may only be directly connected to another neuron within 2 in each dimension. That is to say each neuron has a 5×5 neighborhood of connectivity.

Sparse Arrays
DANNA2 sparse arrays are directed graphs of neurons with fewer connectivity restrictions than grid arrays. For a sparse array to be hardware implementable, it must only have 24 inputs for any one given neuron. However, this can somewhat mitigated with clever usage of fan-in elements.

Many applications will train to higher fitness in less time using a sparse array. Furthermore, sparse arrays typically will require fewer overall neurons, and when deploying a sparse array on an FPGA, unused elements can be optimized out of the design. The net result is much improved effective density and lower power utilization.

Architecture

Accumulate and Fire Neuron

- **Accumulator** - The accumulator is implemented with an adder tree to accommodate all of the possible sources of charge which may arrive in a given cycle. This allows for up to three synapse fires, a leak value, and forwarded charge from a fan-in element to be accumulated into the prior charge in a single cycle.
- **Compare and Fire** - After all the charge has been accumulated for a network cycle, the neuron must compare its charge to its configured threshold. In order to determine if the element should fire, it must also consider the neuron's refractory period. The refractory period is implemented as a loadable counter which loads the configured refractory duration after each outgoing fire. The refractory counter counts down after each network cycle, and the neuron is only permitted to fire when both the refractory counter is zero and the accumulated charge exceeds the threshold. This check occurs on the last sub-cycle count of each network cycle such that the outgoing fire is registered and available at the start of the subsequent network cycle.
- **Charge Forwarding** - The neuron may disable firing and instead be designated as a fan-in element. A fan-in element accumulates charge as a regular element, but rather than comparing against a threshold and firing, it forwards its accumulated charge to a nearby neuron. Each network cycle, the charge starts fresh from a reset state and the immediately prior charge held in the charge forwarding registers.

Synapse

- **Distance Registers** - In order to provide a programmable delay, or distance, for each synapse, incoming fires are fed into the appropriate synapse distance register. Each distance register is a shift register with a configurable tap up to 15 shifts. Each shift register only shifts its data once per network cycle which means it is disabled for nine out of every ten sub-cycles.
- **Synapse Units** - Synapse units are responsible for passing through the synaptic weight to the accumulator if the synapse is firing. When STDP is enabled, the synapse units will track the timing of synapse fires and apply changes to the synapse weight.

Element Control

- **Programming Bus** - In order to configure the parameters and connectivity of each element within the array, each element must allow for a programming interface. In DANNA2, this is implemented with a 48-bit data port and a packet start signal. On the first cycle in which a new packet arrives on this interface, the packet start signal is asserted. The element checks the data to determine if it is the packet's destination. If so, it uses the next 9 cycles of data to configure the element. If the element is not the chosen destination, it forwards the data and the packet start signal to the next element.
- **Programmable Parameters** - These are the values sent through the programming bus in order to represent a given network on a hardware array.

Array Control and Communication

- **AXI-4 Stream** - DANNA2 uses this interface in order to transfer packets to and from other components within the FPGA implementation. DANNA2 actually uses two of these streaming interface. One slave interface is used to receive input packets from a host, and one master interface is used to send output packets to a host.
- **Host to Array Packets** - The host computer may send three types of packets to the DANNA2 hardware array. The host may configure an element's connectivity and parameters with a configuration packet. This is necessary in order to load a new network onto the array. The host may also send fire commands to be executed at a specific cycle with a step and fire packet as shown in figure 5.12. If the host wants to step forward in time but not fire, it may set the network cycle populate each input field with 0 to advance time but not fire. Lastly, the host may choose to reset the state of the array with a reset packet.
- **Array to Host Packets** - Each output fire is arranged in a bit field which allows for up to 416 output neurons to be sent in one packet.

Training

EONS
The primary training method for DANNA2 is the Evolutionary Optimization for Neuromorphic Systems (EONS) platform developed by the TENNLab group. Evolutionary optimization leverages genetic algorithms to determine both the topology and parameters for a neural network. EONS has a generic graph implementation which allows for a model to convert to generic, parameterizable graph. This graph is then transformed using mutation and crossover operations producing a new graph. The model may then convert back to its native format and execute the network. DANNA2 uses this generic graph approach to offload all genetic operations to the current EONS implementation. Depending on DANNA2's own parameters, it can provide EONS with different configurations with varying levels of constraints. EONS will then respect these constraints when generating networks.

3 ShiDianNao

3.1 Introduction

ShiDianNao is a neural network accelerator the is based on output stationary data flow architecture which is specifically optimized for convolutional neural networks (CNN).A ShiDiannao accelerator constructed using a TSMC 65nm technology has a peak performance of 194 GOP/s ,power consumption of 32.0mW and an area consumption of 4.86mm^2.

3.2 Architecture

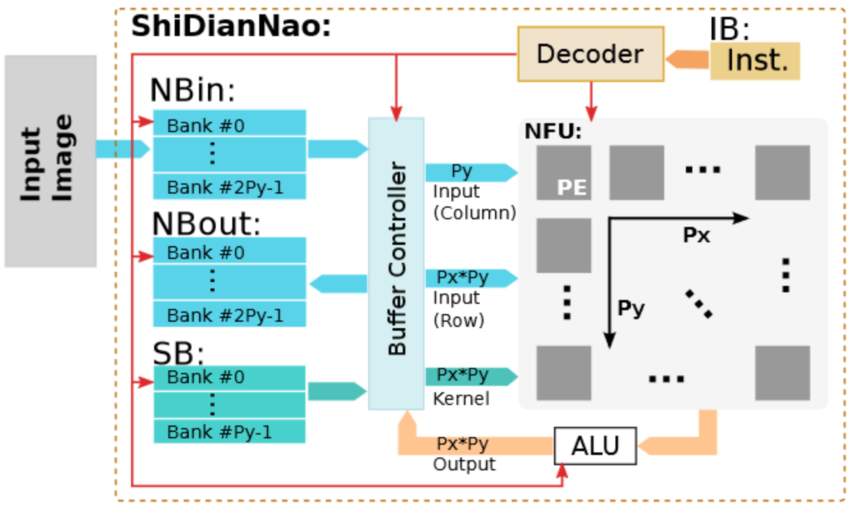


Figure 1: Architecture of ShiDianNao

3.2.1 Neural Functional Unit (NFU)

The NFU structure comprises Px × Py Processing Elements (PEs), aligning naturally with the 2D structure of feature maps.Intra processing element communication is implemented to reduce the band width , thus reducing the complexity in the wiring. Intra processing element communication helps to reduce the required bandwidth by reusing the input given to the processing element on the right and bottom of all the processing elements in the 2D array of processing elements.

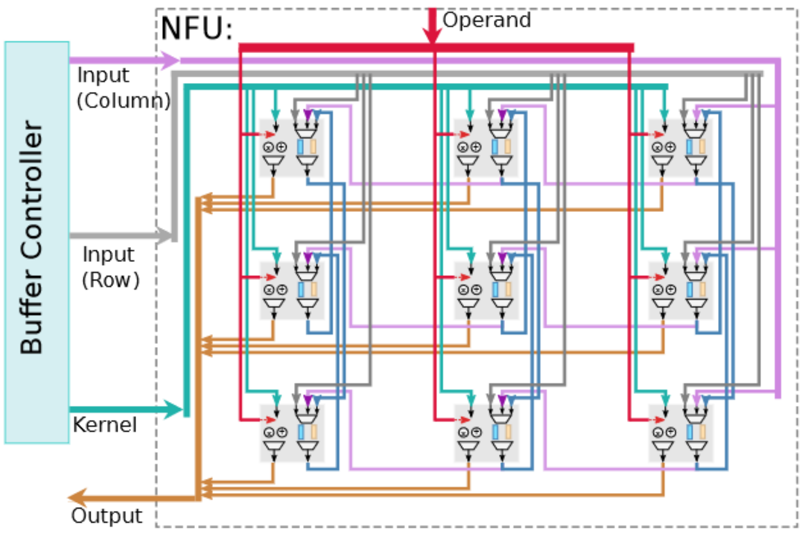


Figure 2: Architecture of Neural Functional Unit (NFU)

- **Processing elements** A processing unit can execute various operations such as multiplication and addition for convolutional, classifier, or normalization layers, solely an addition for an average pooling layer, or a comparison for a max pooling layer, among others, within a single cycle. It possesses three inputs: one for receiving control signals, another for accessing synapses (like kernel values in convolutional layers) from SB, and a third for retrieving neurons from input/output buffers or adjacent processing units. Furthermore, a processing unit features two outputs: one for storing computation outcomes in input/output buffers and another for transmitting locally-stored neurons to neighboring processing units.

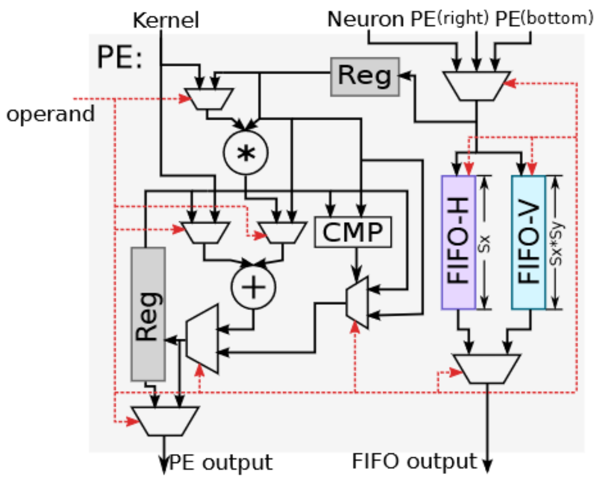


Figure 3: Architecture of the Processing Element

- **Inter-PE data propagation** In CNNs, the output neurons in convolutional, pooling, and normalization layers rely on data from neighboring input neurons within a rectangular window. Typically, these windows significantly overlap for adjacent neurons. However, fetching these overlapping windows for processing repeatedly from the buffer to various processing elements demands substantial bandwidth, potentially causing wiring complications.

To optimize data reuse and alleviate bandwidth issues, we introduce a method for inter-PE data transmission within the PE mesh. This approach enables each processing element (PE) to share locally stored input neurons with its adjacent left and lower neighbors. Implementation involves incorporating two First In First Out registers (FIFOs), namely FIFO-H and FIFO-V, within each processing element. FIFO-H accumulates and transmits data from the input/output buffer (NBin/NBout) and from the right neighbor PE, distributing this information to the left neighbor for reuse. Meanwhile, FIFO-V stores and shares data from NBin/NBout and the upper neighbor PE, disseminating this data to the lower neighbor PE for reuse.

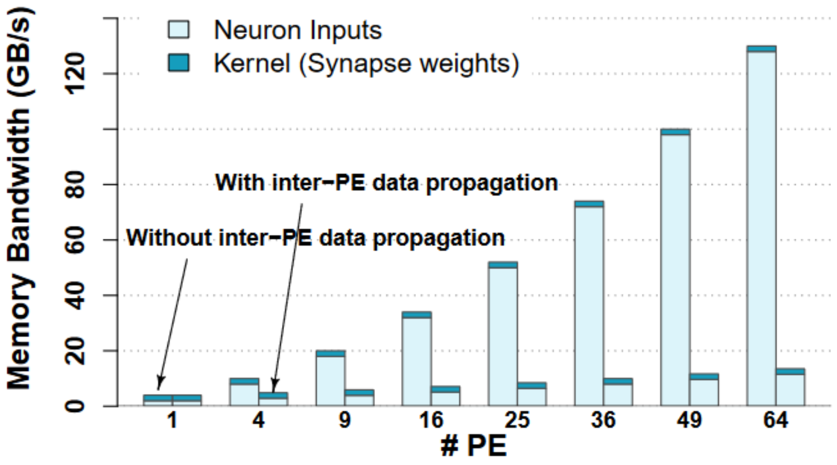


Figure 4: Bandwidth required vs number of processing elements used

Through this inter-PE data propagation mechanism, the internal bandwidth requirements can be significantly diminished, thereby mitigating the demand for high bandwidth within the system

- **Arithmetic Logic Unit (ALU)** A lightweight Arithmetic Logic Unit (ALU) is employed to supplement the processing elements. Within this ALU, various 16-bit fixed-point arithmetic operators are integrated, encompassing division (utilized for average pooling and normalization layers) along with non-linear activation functions such as tanh() and sigmoid() (applied in convolutional and pooling layers). The computation of activation function values is accomplished using piecewise linear interpolation (represented by f(x) = a_ix + b_i, when x exists within the range $[x_i, x_{i+1}]$, where i ranges from 0 to 15). This method is recognized for causing only minimal accuracy loss in Convolutional Neural Networks (CNNs). To facilitate efficient computation, segment coefficients ai and bi are pre-stored in registers, allowing for the approximation to be swiftly computed using a multiplier and an adder.

3.2.2 Storage

An on-chip SRAM is used as the storage for this neural network accelerator which is further divided into NBin to store input neurons , NBout to store output neurons and SB to store intermedite results. The NBin and NBout must be sufficiently large to store the neurons of a whole layer.The NBin and NBout has 2*P_y banks of width 2*P_x bytes each where P_y and P_x are the number of processing elements in a column and a row of the Neural Functional Unit respectively.

3.2.3 Controls

Buffer controllers

On-chip buffer controllers facilitate effective recycling of data and computation within the NFU (Neural Function Unit). For instance, the NB controller, responsible for managing read and write operations in NBin and NBout, efficiently facilitates six read modes and one write mode. These six read modes are employed in various combinations to handle computations required for different layers in a convolutional neural network, such as:

- Read multiple banks from the beginning.
- Read multiple banks from between.
- Read one bank.
- Read a single neuron.
- Read neurons with a given step size.
- Read a single neuron per bank.

Control instructions

Control instructions encompass a set of procedures essential for executing various operations like convolutions, pooling, etc. These instructions are stored in a two-tiered Hierarchical Finite State Machine (HFSM) to outline the execution sequence of the accelerator. Within the HFSM, the first-tier states delineate abstract tasks performed by the accelerator, such as different layer types or ALU tasks. Each first-tier state is associated with multiple second-tier states that define the corresponding detailed low-level execution events. For instance, the second-tier states linked with the first-tier state Conv (convolutional layer) represent phases necessary for processing an input-output feature map pair.

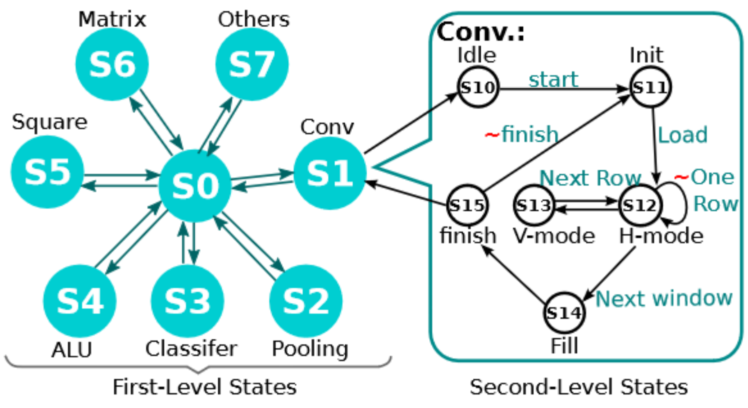


Figure 5: Hierarchical control finite state machine

To represent each HFSM state and its related parameters (e.g., feature map size), a 61-bit instruction is utilized. These instructions can be decoded into precise control signals for a specific number of cycles in the accelerator. This methodology ensures minimal loss of flexibility in practical application. For instance, a CNN requiring 50,000 cycles only demands 1 KB of instruction storage and a lightweight decoder that occupies a mere 0.03 mm^2 in our 65 nm process technology.

3.3 Dataflow

3.3.1 Convolutional layer

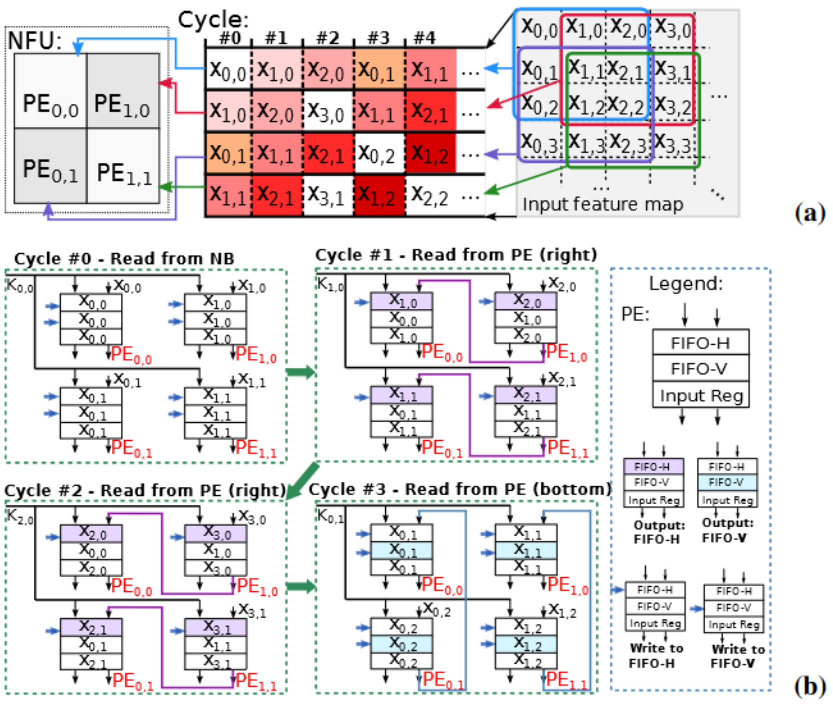


Figure 6: Convolution performed by ShiDianNao

Each individual processing element (PE) within the NFU (Neuron Feature Unit) is dedicated to computing the values for a single output neuron within a convolutional layer. When calculating each output feature map, every PE in the accelerator remains focused on processing a sole output neuron continuously, without switching to another until the current one is computed. The multiplication and accumulation operations for a single output neuron occur sequentially within a series of cycles performed by a single processing element.

Initially, the processing element retrieves data from the Nbin. Subsequently, during the computation of each row of the kernel for the multiply and accumulate operation, the next input neuron is accessed from the FIFO-H (First-In-First-Out buffer for horizontal access) of the processing element to the right. After accessing all the input neurons along a row of the kernel, the subsequent neuron is retrieved from the FIFO-V (First-In-First-Out buffer for vertical access) of the processing element positioned below the currently described processing element. This sequential process continues until all input neurons are obtained and processed.

3.3.2 Pooling layer

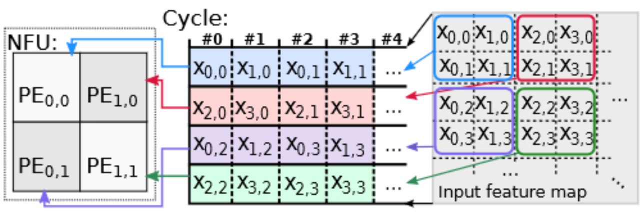


Figure 7: Pooling performed by ShiDianNao

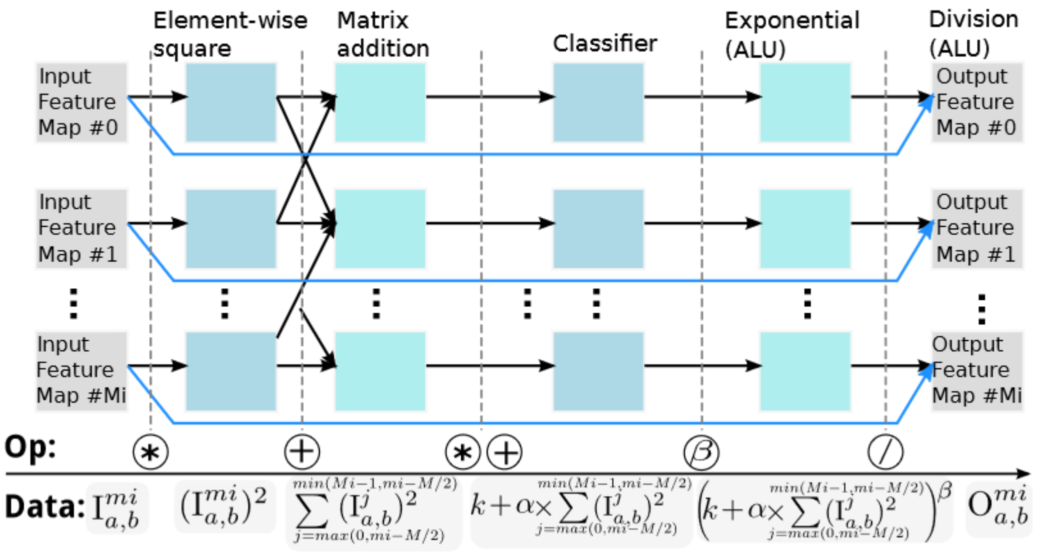
For pooling operation each processing element in the NFU access input neurons only from a single window. As there is no overlap between adjacent windows while pooling in general there is no inter processing element communication. On the cases where there exist overlap between windows inter processing element communication is used.

3.3.3 Classifier layer

In a Convolutional Neural Network (CNN), the convolutional layers enable different pairs of input and output neurons to share synaptic weights using the kernel, while pooling layers do not involve synaptic weights. Conversely, classifier layers typically consist of fully connected neurons without weight sharing among different input-output neuron pairs. Consequently, classifier layers often occupy the most space within the synaptic array, such as 97.28% for LeNet-5. During the execution of a classifier layer, each Processing Element (PE) handles a singular output neuron and remains dedicated to its computation until completion.

In contrast to a convolutional layer, where each cycle involves reading a single synaptic weight and $P_x \times P_y$ different input neurons for all $P_x \times P_y$ PEs, a cycle in a classifier layer requires reading $P_x \times P_y$ distinct synaptic weights and a single input neuron for all $P_x \times P_y$ PEs. Subsequently, each PE performs multiplication between the synaptic weight and input neuron, accumulating the outcome into the partial sum stored in its local register. After several cycles, when the dot product (relating input neurons and synapses) linked with an output neuron is computed, the outcome is forwarded to the Arithmetic Logic Unit (ALU) for the activation function computation.

3.3.4 Normalization Layers



Element-wise square
Matrix addition
Classifier
Exponential (ALU)
Division (ALU)
Input Feature Map #0
Input Feature Map #1
Input Feature Map #Mi
Output Feature Map #0
Output Feature Map #1
Output Feature Map #Mi
Op:
Data:
$\mathrm{I}^{mi}_{a,b}$
$(\mathrm{I}^{mi}_{a,b})^2$
$\sum_{j=max(0,mi-M/2)}^{min(Mi-1,mi-M/2)} (\mathrm{I}^{j}_{a,b})^2$
$k+\alpha\times\sum_{j=max(0,mi-M/2)}^{min(Mi-1,mi-M/2)} (\mathrm{I}^{j}_{a,b})^2$
$\left(k+\alpha\times\sum_{j=max(0,mi-M/2)}^{min(Mi-1,mi-M/2)} (\mathrm{I}^{j}_{a,b})^2\right)^\beta$
$\mathrm{O}^{mi}_{a,b}$

Figure 8: Decomposition of an LRN layer.

Normalization layers can be broken down into several sub-layers and basic computational operations to enable execution by our accelerator. For instance, an LRN (Local Response Normalization) layer is decomposed into various elements including a classifier sub-layer, an element-wise square operation, a matrix addition, exponential functions, and divisions. Similarly, an LCN (Local Contrast Normalization) layer is broken down into multiple components such as convolutional sub-layers, a pooling sub-layer, a classifier sub-layer, matrix additions, element-wise square operations, and divisions.

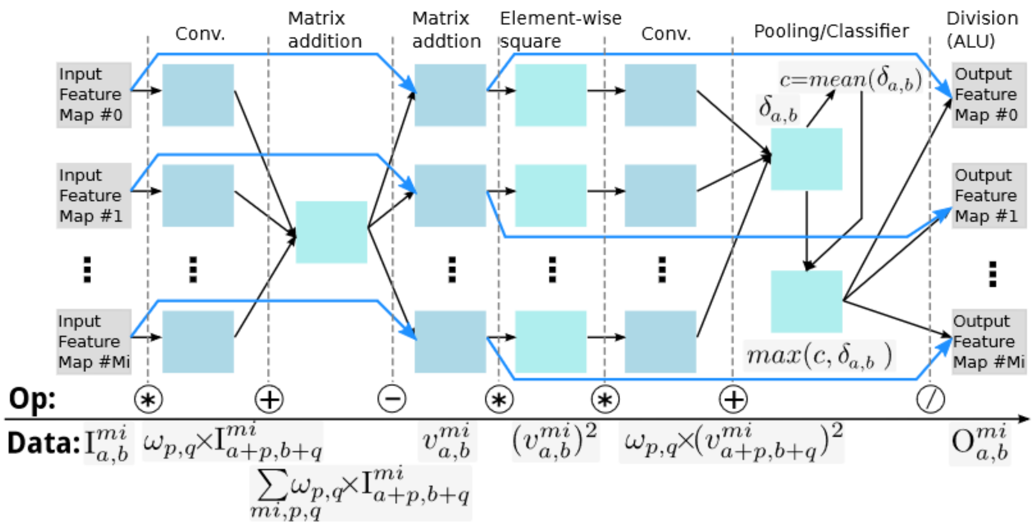


Figure 9: Decomposition of an LCN layer.

Convolutional, pooling, and classifier sub-layers follow rules outlined in earlier sections, while exponential functions and divisions are handled by the Arithmetic Logic Unit (ALU). The remaining computational primitives, like element-wise square operations and matrix additions, are managed by the Neuron Feature Unit (NFU). Supporting these primitives involves each Processing Element (PE) working on a matrix element output using its multiplier or adder in each cycle. Eventually, the results from all Px $\times$ Py PEs are written to NBout.

4 EdgeDRNN

4.1 Introduction

The EdgeDRNN is designed specifically for efficient edge RNN inference at a low latency, particularly with a batch size of 1. It utilizes the delta network algorithm inspired by spiking neural networks to take advantage of the temporal sparsity present in RNNs. This approach involves storing weights in cost-effective DRAM, allowing EdgeDRNN to perform computations for large multi-layer RNNs on budget-friendly FPGA hardware.

Operating at a batch size of 1, EdgeDRNN achieves a mean effective throughput of 20.2 GOp/s and demonstrates a wall plug power efficiency that is more than 4 times higher than that of commercial edge AI platforms.

The primary components of EdgeDRNN include:

- The Delta Unit responsible for encoding delta vectors and generating weight column pointers (pcol).
- The Processing Element (PE) Array designed for matrix-sparse vector multiplications.
- The Control (CTRL) module housing finite state machines (FSMs) and encoding instructions to manage the AXI Datamover.
- Additional modules such as the Configuration (CFG) module composed of configuration registers, the Output Buffer (OBUF) for buffering and redirecting outputs back to the Delta Unit, and the W-FIFO for buffering weights.

4.2 Arcihtecture

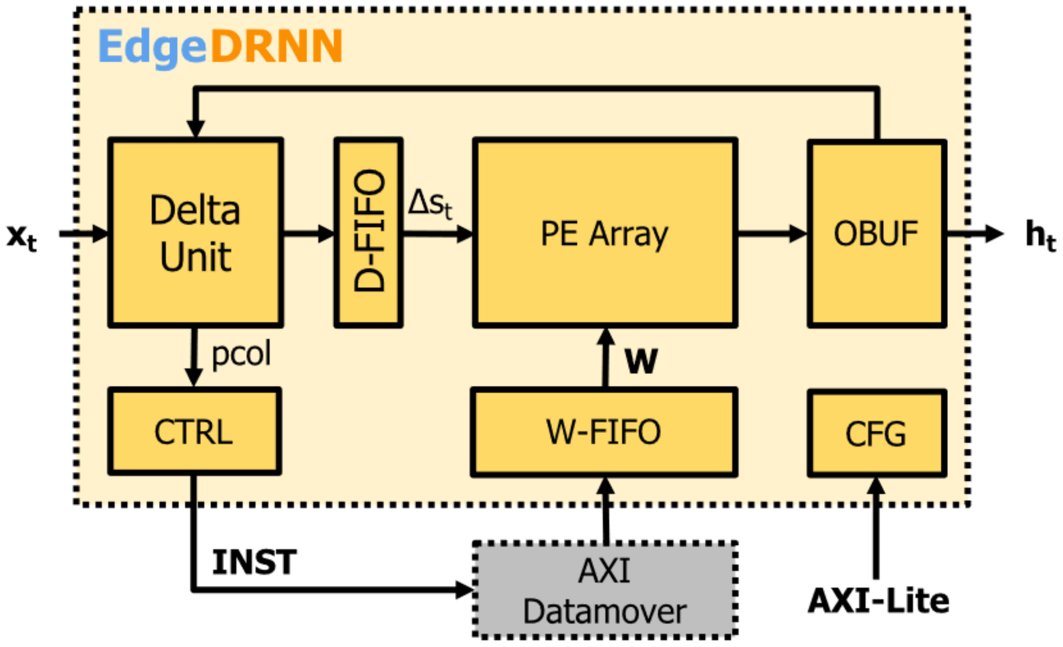


Figure 10: Architecture of EdgeDRNN.

4.2.1 Delta Unit

The Delta unit serves the purpose of computing alterations in the input vector (x_t) and the prior neuron activation (h_{t-1}) concerning their preceding values. It operates based on the state of a finite state machine, selecting either (x_t) or (h_{t-1}) for processing, handling one element of the chosen vector's delta state per cycle. Upon processing an element, the change in state is compared to a threshold value. If this change surpasses the threshold, the new delta state vector elements meeting or exceeding the threshold and their respective physical weight column addresses (pcol) are sent to the D-FIFO and CTRL. Simultaneously, the corresponding state element (st) is updated in the BRAM to modify the state memory. Conversely, if the elements fall below the threshold, they are disregarded and not added to the D-FIFO. To minimize latency, employing multiple Delta units (N units) enables the processing of subsections of the state elements (st) in segments of length N.

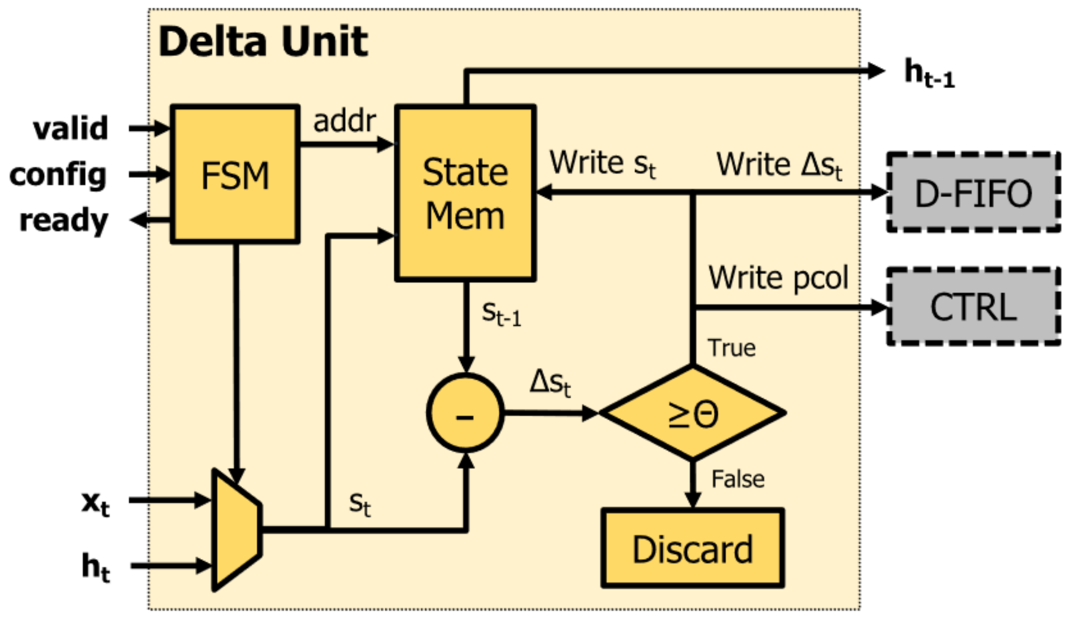


Figure 11: Architecture of delta unit.

4.2.2 CTRL unit

The CTRL module contains FSMs that control the PE array. This module generates 80-bit instructions for controlling the AXI Datamover to fetch RNN parameters. The instruction contains pcol and the burst length calculated from the dimensions of the network stored in configuration registers.

4.2.3 Processing Element Array

The Processing Element (PE) consists of a 16-bit multiplier (MUL) and two adders: a 32-bit ADD0 and a 16-bit ADD1. To enable versatility for both MxV and vector dot products, multiplexers are positioned before the MUL's operands. ADD0's multiplexer allows the selection between '0' and BRAM data, opting for '0' during BRAM initialization. ADD1 manages element-wise vector additions. All these units are parameterized in System Verilog RTL, allowing configuration during compile-time to support any fixed-point precision within their designated bit width. Additionally, the PE facilitates tanh and sigmoid functions through lookup tables (LUTs), with a fixed 16-bit input bit width for the LUTs and an adjustable output bit width ranging from 5 to 9 bits.

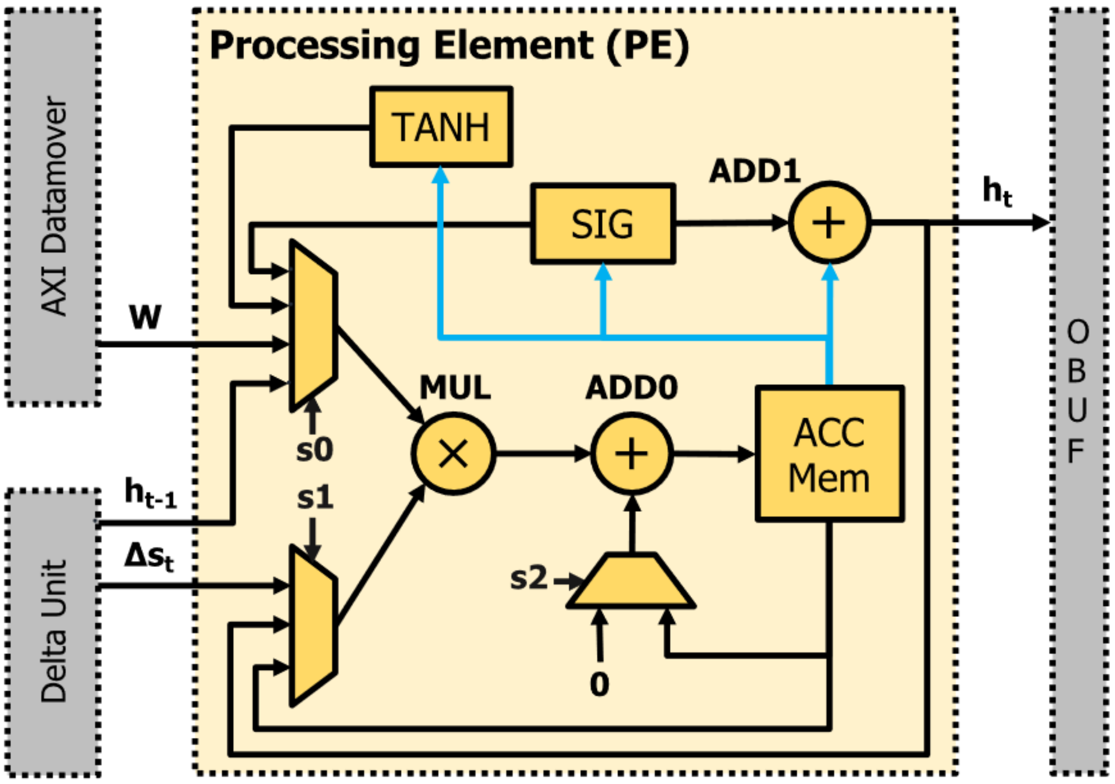


Figure 12: Architecture of processing element.

4.3 Dataflow

The Delta Unit and the control unit work together to estimate key parameters, including delta vectors and physical weight column pointers. These parameters are then passed on to the D-FIFO and CTRL modules for further processing. The GRU-RNN's weight matrices are combined, and biases are added as the first column. Additionally, an element of 1 is appended to each input state vector as the first element.

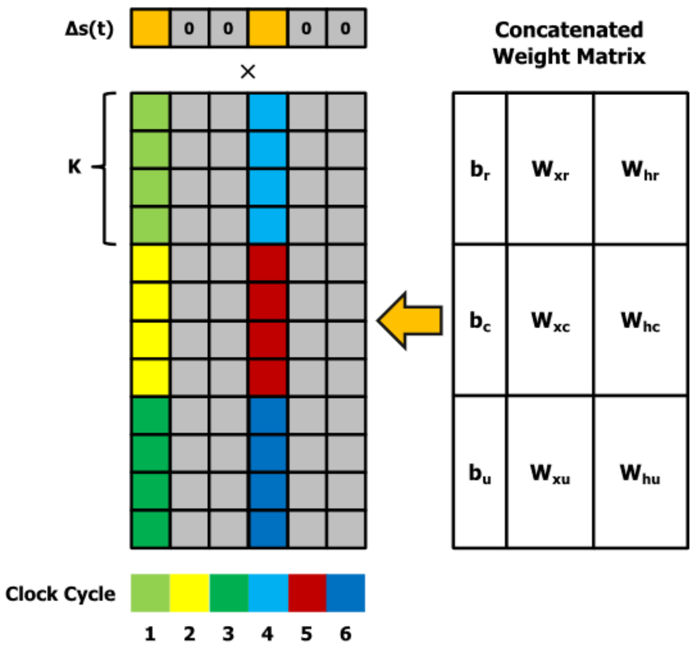


Figure 13: Dataflow of matrix vector multiplication (MxV).

The Processing Array (PE array) plays a crucial role in the efficiency of EdgeDRNN. It performs selective multiplication, only processing non-zero delta state elements with their corresponding valid columns. The products are accumulated in the Accumulation Memory (ACC Mem) to compute delta memory vectors.

Furthermore, the PE array is responsible for calculating the activation state h_t after the matrix-vector multiplication (MxV). It fetches the delta memory vectors from the ACC Mem and utilizes an 8-stage pipeline to calculate h_t. Flip-flops are employed to buffer paths without any operator in any stage for one clock cycle.

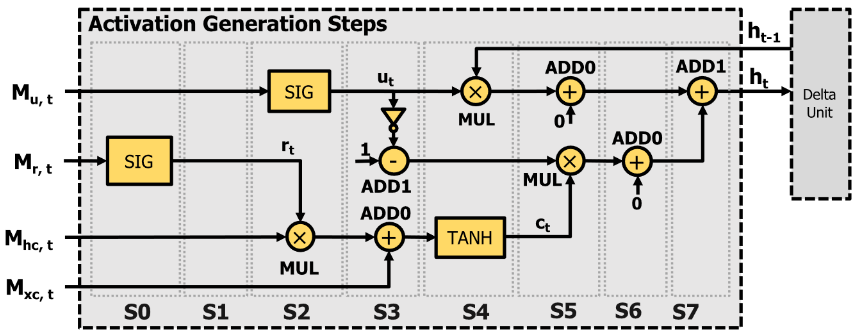
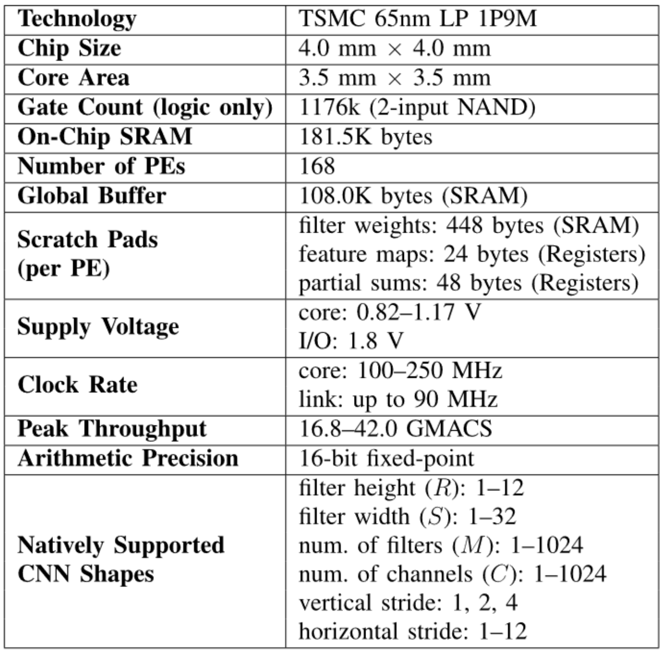


Figure 14: Stages of GRU activation pipeline in the PE Array.

5 Eyeriss

5.1 Introduction

Eyeriss is an accelerator for state-of-the-art deep convolutional neural networks (CNNs). It optimizes for the energy efficiency of the entire system, including the accelerator chip and off-chip DRAM, for various CNN shapes by reconfiguring the architecture. Eyeriss uses a row stationary dataflow with 168 processing elements. Row stationary dataflow reconfigures the computation mapping of a given shape, which optimizes energy efficiency by maximally reusing data locally to reduce expensive data movement, such as DRAM accesses. Compression and data gating are also applied to further improve energy efficiency. Eyeriss processes the convolutional layers at 35 frames/s and 0.0029 DRAM access/multiply and accumulation (MAC) for AlexNet at 278 mW (batch size N = 4), and 0.7 frames/s and 0.0035 DRAM access/MAC for VGG-16 at 236 mW (batch size N = 3).



Technology	TSMC 65nm LP 1P9M
Chip Size	4.0 mm × 4.0 mm
Core Area	3.5 mm × 3.5 mm
Gate Count (logic only)	1176k (2-input NAND)
On-Chip SRAM	181.5K bytes
Number of PEs	168
Global Buffer	108.0K bytes (SRAM)
Scratch Pads (per PE)	filter weights: 448 bytes (SRAM) feature maps: 24 bytes (Registers) partial sums: 48 bytes (Registers)
Supply Voltage	core: 0.82–1.17 V I/O: 1.8 V
Clock Rate	core: 100–250 MHz link: up to 90 MHz
Peak Throughput	16.8–42.0 GMACS
Arithmetic Precision	16-bit fixed-point
Natively Supported CNN Shapes	filter height (R): 1–12 filter width (S): 1–32 num. of filters (M): 1–1024 num. of channels (C): 1–1024 vertical stride: 1, 2, 4 horizontal stride: 1–12

Figure 15: Specifications of Eyeriss.

5.2 Architecture

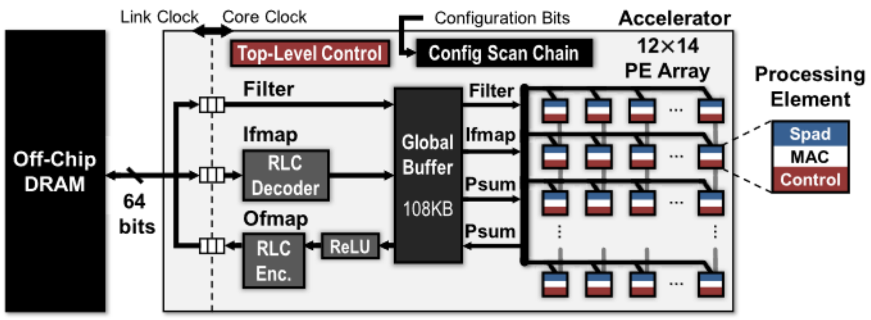


Figure 16: Architecture of Eyeriss.

5.2.1 Storage

Global buffer (GLB)

The purpose of the GLB is to store the input feature maps (ifmap) , filter weights and the output feature maps (ofmap). While the PE array is working on a processing pass, the GLB preloads the filters used for the next processing pass.

Spads

These are the small , high speed storage units that is present inside a processing element. The purpose of this buffer is to store the row of a the filter of weights and the row of the input feature map to compute the 1D convolution primitives.

5.2.2 Processing element

The architetcture of Processing Element is as shown. First In First Out storage buffers are used at the I/O of each Processing element to balance the workload between the NoC and the computation. The datapath is pipelined into three stages: one stage for spad access, and the remaining two for computation. The computation consists of a 16-bit two-stage pipelined multiplier and adder. Since the multiplication results are truncated from 32 to 16 bit, the selection of 16 bit out of the 32 bit is configurable, and can be decided by the dynamic range of a layer from offline experiments. Spads are separated for three data types to provide enough access bandwidth.

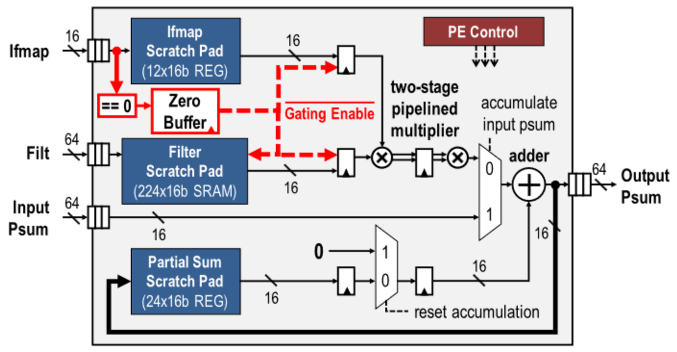


Figure 17: Architecture of the processing element.

Data gating logic is implemented to exploit zeros in the input feature map for saving processing power. An extra 12-bit Zero Buffer is used to record the position of zeros in the input feature map spad. If a zero input feature map value is detected from the zero buffer, the gating logic will disable the read of the filter spad and prevent the MAC datapath from switching. Compared with the PE design without the data gating logic, it can save the PE power consumption by 45%.

5.2.3 Network on Chip (NoC)

The NoC manages data delivery between the GLB and the PE array as well as between different PEs. The NoC used here has 3 trees and they are as follows

- **The Global Input Network (GIN)** efficiently sends data from the Global Buffer (GLB) to Processing Elements (PEs) requiring the same information (filter weights, input features, or partial sums) within a single cycle. It utilizes a two-level hierarchy: Y-buses and X-buses. Each Y-bus connects to 12 X-buses (one per PE row) that, in turn, connect to 14 PEs in that row. Both X-buses and PEs have reconfigurable IDs, allowing for dynamic grouping based on data requirements in each layer. Data read from the GLB is tagged with (row, column) information by the controller, ensuring delivery to the intended X-bus and PEs based on ID matching within a single cycle. This matching is performed by Multicast Controllers (MCs) on both Y and X-buses. Unmatched X-buses and PEs are disabled to conserve energy. To prevent data loss, data transmission from the GLB to the GIN only occurs when all PEs are ready to receive it. Eyeriss employs separate GINs for filter weights, input features, and partial sums to provide sufficient bandwidth. Each GIN uses 4-bit row IDs for the 12 rows. Filter and partial sum GINs use 4-bit column IDs for the 14 columns, while the input feature map GIN uses 5 bits to accommodate up to 32 input feature map rows transmitted diagonally. The data bus width is 64 bits (4 × 16 bits) for filter and partial sum GINs and 16 bits for the input feature map GIN.

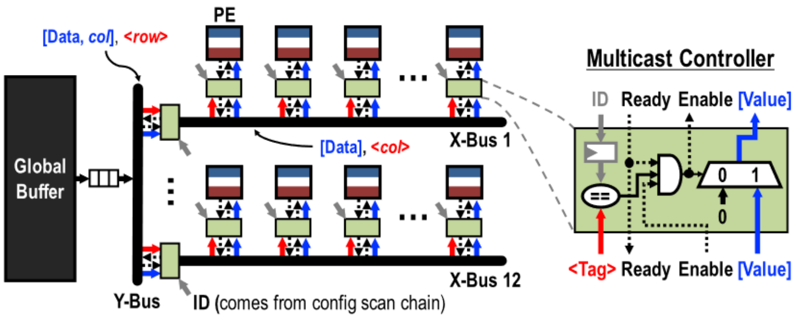


Figure 18: Architecture of Global Input Network (GIN).

- **Global Output Network (GON)** GON is used to read the partial sums generated by a processing pass from the PE array back to the GLB. The GON has the same architecture as the GIN; only the direction of data transfer is reversed. The data bus width is also 64bit as the partial sums GIN.

- **Local Network (LN)** Between every pair of PEs that are on two consecutive rows of the same column, a dedicated 64b data bus is implemented to pass the partial sums from the bottom PE to the top PE directly. Therefore, a PE can get its input partial sums either from the partial sums GIN or LN. The selection is static within a layer, which is controlled by the scan chain configuration bits and only depends on the dataflow mapping of the CNN shape.

5.2.4 Dataflow

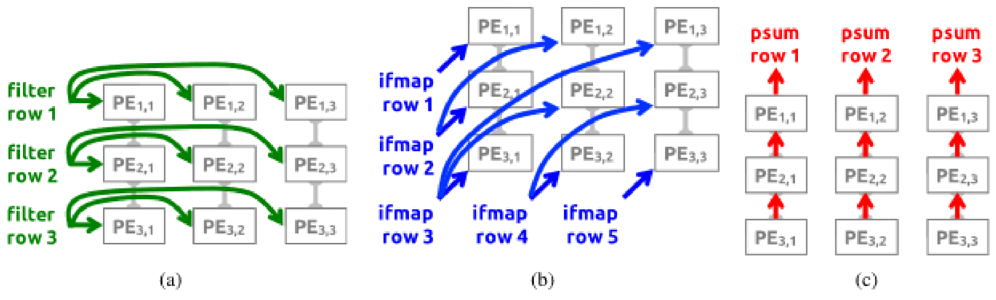


Figure 19: Dataflow in eyeriss.

Eyeriss uses Row Stationary dataflow to efficiently reuse the filter weights , data in the input feature map and the output feature map. The processing elements are arranged as a grid. The filter rows are passed horizontally , so that each processing element in a row receives the filter weights inside a single row of the filter. The rows of the input feature map is passed diagonally in which each processing along the off diagonal receives the values present in 1 row of input feature map. The partial sums accumulated on all the processing elements along a column is summed together to get the value of a single neuron in the output feature map.

6 Heterogeneous Dataflow Accelerators

The data flow accelerators refers to the hardware that are used to accelerate the computing and are classified as follows:

- **Fixed Dataflow Accelerators :** These are the dataflow accelerators that has a single architecture. These kind of accelerators would perform well for a few types of layers but is not suitable for other kind of layers.
- **Reconfigurable Dataflow Accelerators :** These ASIC accelerators follow a coarse-grained reconfigurable architecture (CGRA) style. An essential issue concerning these Reconfigurable Dataflow Accelerators (RDAs) is that while they offer flexibility, this flexibility comes with an added expense in terms of hardware components such as switches and wires. This factor becomes worrisome when considering their implementation in edge, mobile, and cloud devices where strict energy constraints exist. For instance, in our evaluation, MAERI consumed, on average, 11.7% more energy compared to an NVDLA-style Flexible Dataflow Architecture (FDA).
- **Heterogeneous Dataflow Accelerators :** These are accelerators designed based on dataflow principles, offering adaptability through the use of several sub-accelerators, each specialized for a distinct dataflow, all integrated into a single chip. HDAs offer two crucial functionalities:
  - **Dataflow flexibility**, enabled by scheduling each layer from the multiple DNN models on the most efficient sub-accelerator for each layer, as long as possible.
  - **High utilization**, enabled by scheduling multiple layers from different models across the sub-accelerators simultaneously

6.1 Significance of Heterogeneous Dataflow Accelerators

To attain superior outcomes in tasks such as classification, prediction, and more, numerous applications currently utilize Deep Neural Networks (DNNs) as their core framework. These DNNs are employed for various functions like face recognition, image segmentation, depth estimation, among others. As newer applications such as Augmented Reality/Virtual Reality (AR/VR) combine DNNs for intricate tasks, they generate multi-DNN workloads. These individual DNNs designed for different sub-tasks exhibit considerable diversity. This diversity in models results in notable differences in layer configurations and operations, creating a need for Heterogeneous Dataflow Accelerators.

6.2 Design Considerations and Definition of HDA

Designing a Heterogeneous Dataflow Accelerators can be divided in to the following 3 steps

- **Dataflow Selection for Sub-accelerators :** Selecting appropriate dataflow styles for constructing sub-accelerators within an Heterogeneous Dataflow Accelerator (HDA) significantly influences its overall efficiency. To fully leverage the advantages of dataflow flexibility, the dataflow styles employed in these sub-accelerators must possess considerable diversity. This diversity ensures that the resulting HDA can effectively accommodate various layers characterized by distinct shapes and operations.
- **Hardware Resource Partitioning :** Allocating hardware resources uniformly among sub-accelerators frequently results in less-than-optimal design configurations for an Heterogeneous Dataflow Accelerator (HDA). Therefore, to achieve the most effective hardware partitioning, a Design Space Exploration (DSE) algorithm is utilized. This approach aims to identify the best possible configuration within the hardware design space.
- **Layer Scheduling :** As HDAs consist of multiple instances of accelerators, it's crucial to establish the sequence for executing layers across these sub-accelerators, a process vital for leveraging layer parallelism. A scheduler is necessary to verify whether the schedules generated adhere to layer dependencies and memory limitations. Furthermore, this scheduler must aim to optimize both overall latency and energy consumption, considering the holistic efficiency of the entire HDA, rather than focusing solely on individual sub-accelerator. optimizations.

6.3 Benefits of Heterogeneous Dataflow Accelerators

The benefits of HDAs are as follows :

- **Selective Scheduling** As each layer prefers different dataflow and hardware, running each layer on its most preferred sub-accelerator in an HDA is an effective solution to maximize overall efficiency.
- **Layer Parallelism**Unlike most FDAs and RDAs that run one layer and another, HDAs can simultaneously run multiple layers of different models. By this approach, an HDA can overlap the latency of multiple models, which leads to latency hiding among DNN models reducing overall latency.
- **Low Hardware Cost for Dataflow Flexibility** As HDA employs FDA style sub-accelerators, HDAs do not involve the costs for reconfigurability like RDAs.