electronics

MDPI

Article

Implementation of Autoencoders with Systolic Arrays through OpenCL

Rafael Gadea-Gironés *,†, Vicente Herrero-Bosch †, Jose Monzó-Ferrer † and Ricardo Colom-Palero †

Institute for Molecular Imaging Technologies (I3M), Universitat Politècnica de València, 46022 Valencia, Spain; viherbos@eln.upv.es (V.H.-B.); jmonfer@upvnet.upv.es (J.M.-F.); rcolom@eln.upv.es (R.C.-P.)

* Correspondence: rgadea@eln.upv.es

† These authors contributed equally to this work.

Abstract: In the world of algorithm acceleration and the implementation of deep neural networks' recall phase, OpenCL based solutions have a clear tendency to produce perfectly adapted kernels in graphic processor unit (GPU) architectures. However, they fail to obtain the same results when applied to field-programmable gate array (FPGA) based architectures. This situation, along with an enormous advance in new GPU architectures, makes it unfeasible to defend an acceleration solution based on FPGA, even in terms of energy efficiency. Our goal in this paper is to demonstrate that multikernel structures can be written based on classic systolic arrays in OpenCL, trying to extract the most advanced features of FPGAs without having to resort to traditional FPGA development using lower level hardware description languages (HDLs) such as Verilog or VHDL. This OpenCL methodology is based on the intensive use of channels (IntelFPGA extension of OpenCL) for the communication of both data and control and on the refinement of the OpenCL libraries using register transfer logic (RTL) code to improve the performance of the implementation of the base and activation functions of the neurons and, above all, to reflect the importance of adequate communication between the layers when implementing neuronal networks.

Keywords: OpenCL; neural networks; systolic arrays; FPGA

check for updates

Citation: Gadea-Gironés, R.; Herrero-Bosch, V.; Monzó-Ferrer, J.; Colom-Palero, R. Implementation of Autoencoders with Systolic Arrays through OpenCL. *Electronics* **2021**, *10*, 70. https://doi.org/10.3390/electronics10010070

Received: 1 December 2020
Accepted: 31 December 2020
Published: 3 January 2021

1. Introduction

Currently, the field of learning machines and, more specifically, of deep neural networks is currently rife with acceleration environments. Those try to bring electronic technologies of a certain complexity closer to engineers and researchers who make use of powerful and versatile artificial intelligence (AI) frameworks such as Tensorflow, Caffe, and PyTorch.

These environments are usually oriented toward the determination of the network topology and its subsequent training with learning samples. However, when you want to use that trained network in the inference phase, the hardware that carries out the training phase required to create this artificial intelligence is hardly adequate due to size, price, or power limitations. An evaluation of different available electronic technologies is required to efficiently perform these tasks. This is the point where different implementation options such as application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs) are able to provide competitive solutions compared to today's predominant central processing units (CPUs) and graphical processing units (GPUs). As a consequence, a second wave of applications and languages in search of fast implementation and optimal results has appeared. These applications focus on the inference phase of artificial neural networks. Although they communicate easily with training environments and allow the heterogeneity of technological alternatives (CPU, GPU, FPGA), they lack architectural vision when dedicated to FPGA solutions. For instance, OpenVINO and the Xilinx Machine Learning Suite provide direct implementation for FPGA and GPU based solutions, but on the other hand, oneAPI aims at providing a new language for machine learning hardware design.

Finally, the goal of this work is to find a middle abstraction point of implementation for deep neural networks. A trade off is made between designing with the applications described above and the classic microelectronic workflow based on hardware description languages (HDLs, such as VHDL and Verilog) and register transfer logic (RTL) synthesizers. We will focus first on the implementation of autoencoders, which are the simplest example of dense layer stacking. Autoencoders rely on unsupervised training, which enables them to automatically learn from data examples. This implies that it is easy to train specialized instances of the algorithm that will match a specific type of input and will not require any new engineering but the appropriate training data. Its features for data denoising and dimensionality reduction [1–5] are very useful to us for our data streams of signals acquired for medical imaging instrumentation.

This middle point approach is required since it is necessary to accelerate the microelectronic design beyond the possibilities of RTL synthesis and also make it more accessible to systems engineers. However, from our point of view, in order to improve the performance of the tools that try to automate the entire process, the architectural control of the obtained implementations must be guaranteed. This new methodological solution must satisfy the temporary requirements of our applications dedicated to medical imaging, with very demanding throughputs below 10 µs. These kinds of requirements are the ones that have mainly motivated this new way of approaching the inference phase of artificial neural networks proposed by this paper.

The rest of this paper is organized as follows: The machine learning design infrastructures currently used with FPGA as the target technology are reviewed in Section 2. In Section 3, the elements used in the proposed methodology are determined. In Section 4, the obtained implementation results for an autoencoder are analyzed, comparing them with other available environments. The conclusions are presented in Section 5.

2. Autoencoder Implementation Review

Since few pure autoencoder implementations have been found in the bibliography [6–8], we will focus on networks with the same characteristics as autoencoders. A sufficient number of implementations were selected to draw some conclusions about our starting hypothesis on the trade off between high level synthesis (HLS) and register transfer logic (RTL) with HDLs.

The set of neural networks under study includes multilayer perceptrons and convolutional neural networks (CNNs). Many implementations of the former have been described and developed in the past, but currently, the latter draws all the attention of the different research groups working in this field using FPGAs.

Extensive CNN implementations on FPGAs were fully covered in [9,10]. The first review reported up to 30 different implementations, 18 of which were based on HLS and 9 on RTL synthesis. Unfortunately, there are no data about the design time of such implementations since research articles do not usually include it, and it is practically impossible to compare the performance due to the lack of a uniform example since they work with different precision levels. For example, the concept of GOPS/W (giga operations per second/Watt) is meaningless due to of the differences in the computed operations.

The second review was chosen based on the use of dedicated training tools such as Caffe, Tensorflow, PyTorch, and Theano. These packages allow fast implementations of CNN on different hardware platforms. Since the results obtained with these tools are quite heterogeneous, the review aimed at throughput and latency when evaluating their quality.

A detailed examination of the aforementioned reviews sheds light on the usage of an intermediate level of implementation. The meet in the middle approach shown in [11,12] is a trade off between high level and RTL synthesis workflows. A combination of OpenCL and systolic architectures was also shown in [12].

Regarding autoencoder implementations, HLS based ones [6] and others directly obtained from HDL were shown in [7,8,13].

As expected, HLS implementations are flexible and may also include a training phase. However, from a performance point of view, such implementations are only slightly optimized. Their throughput measured in terms of frames per second (fps) is very low. In [6], a deep network implementation based on stacked sparse autoencoders was discussed. This implementation is a multilayer perceptron-type network created by stacking trained autoencoders without supervision; thus, a 3072-2000-750-10 topology network is constructed (3072 nodes in the first layer, 2000 nodes in the first hidden layer, 750 nodes in the second hidden layer, and 10 nodes in the output layer).

HDL based implementations are more efficient, improving throughput by two orders of magnitude. However, the design process is complex and has a limited flexibility. In [7,8,13], proposals for traditional neural networks using autoencoders were introduced. Suzuki et al. [7] proposed, in addition to the traditional autoencoder architectures, a structure with two chained autoencoders (4-2-1-2-4) inside a Xilinx Virtex-6 XC6VLX240T. Medus et al. [13] proposed a systolic architecture for different neural networks, highlighting an autoencoder implementation (784-196-784) in fixed-point arithmetic, with 3.24 µs of throughput inside a Xilinx Virtex-7 XC7VX485T. In [8], a hardware implementation proposal of a deep neural network based on the stacked sparse autoencoder technique was presented. The hardware was developed for the feedforward phase, adopting the systolic array technique, which allowed them to use multiple neurons and several layers. Data regarding hardware occupancy rate and processing time were presented for a Xilinx Virtex-6 XC6VLX240T-1FF1156 FPGA. This implementation can reach throughputs of 26,000 images/s.

In this paper, the proposed implementation is an attempt to combine the virtues of both trends. We aim at exploiting OpenCL's flexibility for describing systolic architectures. The latter ones are implemented using multikernel structures of the single-work-item type. Each kernel is tuned with SystemVerilog based RTL libraries, which were added to the OpenCL compiler to improve the performance in two fundamental aspects: the implementation of activation functions and better usage of the ARRIA 10 architecture's floating point digital signal processors (DSPs), especially when they execute the $Accu = A \times B + Accu$ equation.

The main underlying idea of a trade off between HLS and RTL has already been proposed in previous works [11], and the use of systolic architectures was also fundamental in [12], although it was performed in a less customized way.

3. Methodologies Used

Methodologically, the following aspects must be described:

1. The generation of the systolic architectures: from this point of view, a classic systolization system is used;
2. The transfer of the obtained architecture to OpenCL code; and
3. Kernel implementation refinement using RTL libraries.

Fundamental contributions in these three areas have already been provided by our group, as follows:

1. Systolic generation:
    - The most important contribution is the implementation of the dense layers by pairs, not individually. An autoencoder is a perfect choice for demonstrating this idea, since it is made of two dense layers named the encoder and decoder.
    - Projections are needed in order to allow several processing elements (PEs) to be executed by the same kernel. This fact will help in saving resources in the hardware implementation of the systolic array generation. The best projections are chosen based on their efficiency in layer-to-layer communication.
2. Conversion of the systolic architecture to OpenCL:

- The use of a multikernel implementation, in which the projected processing units are assumed by a single-work-item kernel with autorun; this element will be referred to as the projected processing element (PPE);
- The use of kernels for data entry and data extraction based on two techniques: buffers and queues;
- The use of two control techniques: distributed control allocated inside the processing unit kernel or centralized control;
- The use of control flow and channel data flow, which will play a key role in the synchronization of all processing units.

3. RTL refinement:
   - The implementation of the activation functions by means of embedded memories; and
   - The implementation by direct instantiation of the operations' output = $A \times B + C$ and $Accu = A \times B + Accu$ using the embedded DSPs.

3.1. Systolic Generation

A systematic method was applied for the design of systolic arrays [14] in order to obtain the architecture of two adjacent dense layers. This method is suitable for algorithms that can be expressed as a set of uniform recurrent equations on a convex set of whole D coordinates [14]. Once the recurring equations have been obtained, the method follows two steps: (1) finding a plan for the operations (schedule) that is compatible with the dependencies introduced by the equations and, then, (2) mapping the D domain within another finite set of coordinates. Each of these coordinates will represent a processor of the systolic array. As a consequence, the concurrent operations are mapped within different processors (allocation). The schedule and allocation functions must fulfill conditions that allow the method to be fully automated. This fact enables the transition to a systolic array of equations that intervene in the operation of the multilayer perceptron.

One of the main issues when designing neural network architectures is the ability to share memory, hardware resources, and the array configuration in both training mode and already-trained work mode ("recall" phase) computations. This fact avoids the difficult task of transferring synaptic weights between the recall and training phases, carrying the same memory elements used to store weights along all the computations. Our example makes use of backpropagation training algorithm.

The resulting architecture was obtained using different procedures to the ones shown in [15–17]. The latter procedures start from the dependency network and conduct a mapping operation on a systolic array. Projection operations in a certain direction are carried out for each PPE. The proposed architecture is called "alternative orthogonal systolic array", and the complete method of extraction can be seen in [18], though it is not systematic and therefore difficult to transfer to other cases. In the case presented herein, the systematic procedure seen in [14] is used. Even though the same configuration already obtained by Murtagh, Tsoi, and Bergmann is achieved here, this methodology can be executed for the computation of different configurations and can be applied to other types of problems (algorithms).

3.1.1. Encoder

An autoencoder with the same layer structure as the one shown in Figure 1 is used.

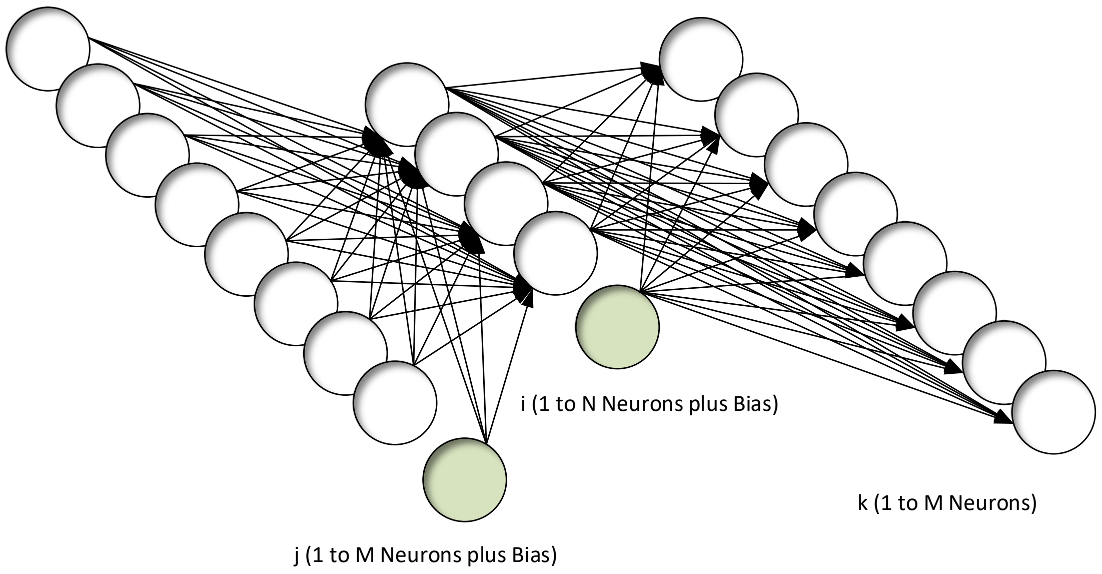


Figure 1. Autoencoder.

Domain Definition

Here, the encoder layer connection between *j* (neuron source layer) and *i* (neuron destination layer) is discussed (Figure 2).

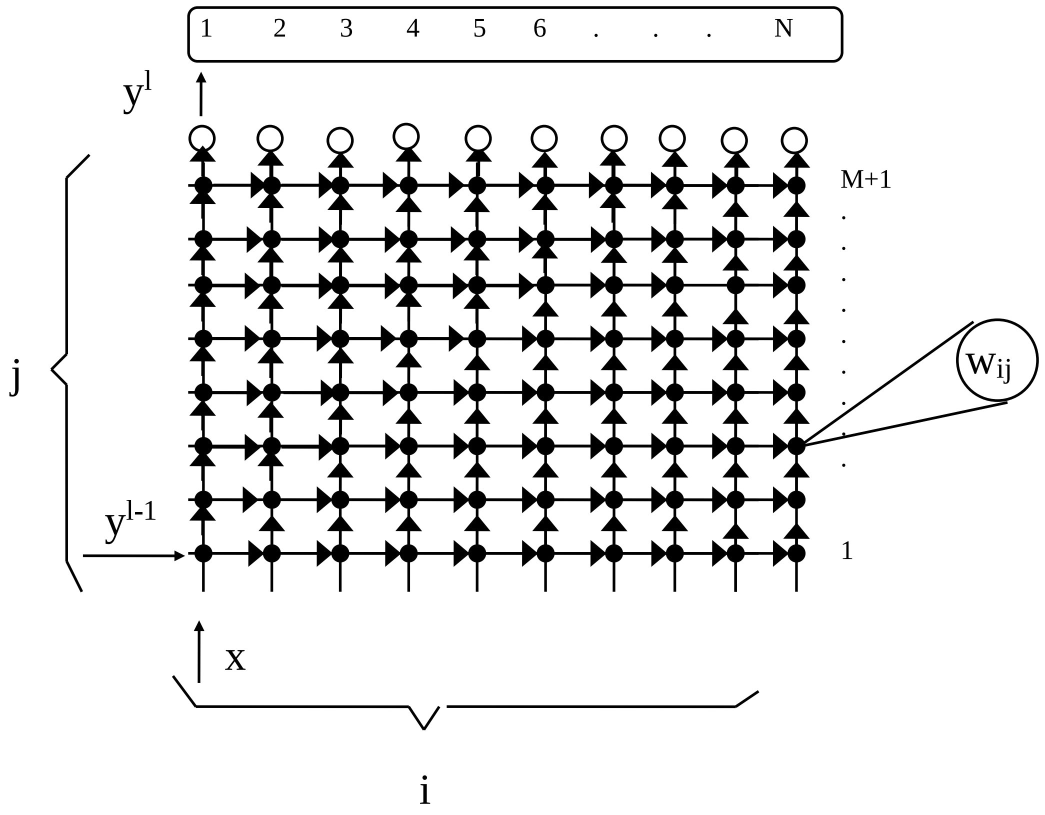


Figure 2. Encoder layer interconnection.

1. Uniform Recurrence Equations

From Equations (1) and (2), uniform iterative Equations (3) and (4) can be obtained:

$$u_i^l = \sum_{j=0}^{M+1} w_{ij}^l y_j^{l-1} \tag{1}$$

$$y_i^l = f(u_i^l) \tag{2}$$

$$1 \le i \le N,\ 1 \le l \le L,$$

where y stands for activation, w for weight, f for the non-linear function, and j, i for the indexes in the source and destination layer.

$$layer\ l \left\{ \begin{array}{l} x^l(i,j) = x^l(i,j-1) + w^l(i,j) \cdot y^{l-1}(i-1,j) \\ y^{l-1}(i,j) = y^{l-1}(i-1,j) \\ y^l(i,M+2) = f(x^l(i,M+1)) \\ 1 \le i \le N \\ 1 \le j \le M+1 \end{array} \right\}, \tag{3}$$

and the boundary conditions:

$$layer\ l \left\{ \begin{array}{l} y^{l-1}(0,j) = f(x^{l-1}(j,L+1)) \quad \text{for} \quad \text{j} = 1 \ldots M \\ y^{l-1}(0,M+1) = 1 \qquad \text{Bias} \\ x^l(i,0) = 0 \quad \text{for} \quad \text{i} = 1 \ldots N \end{array} \right\}. \tag{4}$$

2. Δ = dependence vectors = { $(0,1)_{d_1}$ $(1,0)_{d_2}$}
3. D = {(i,j) $N \ge$ i ≥ 1 $M+1 \ge j \ge 1$ }
4. Domain geometry.

The geometry of the domain is shown in Figure 3.

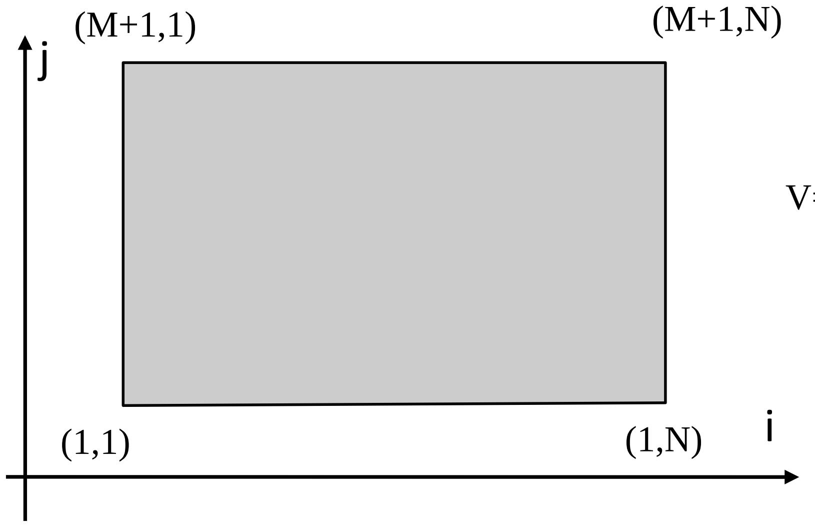


Figure 3. Domain geometry in the encoder layer.

Resulting Systolic Array

The final result of the systolization procedure with a direction (1,0) projection is the systolic array shown in Figure 4.

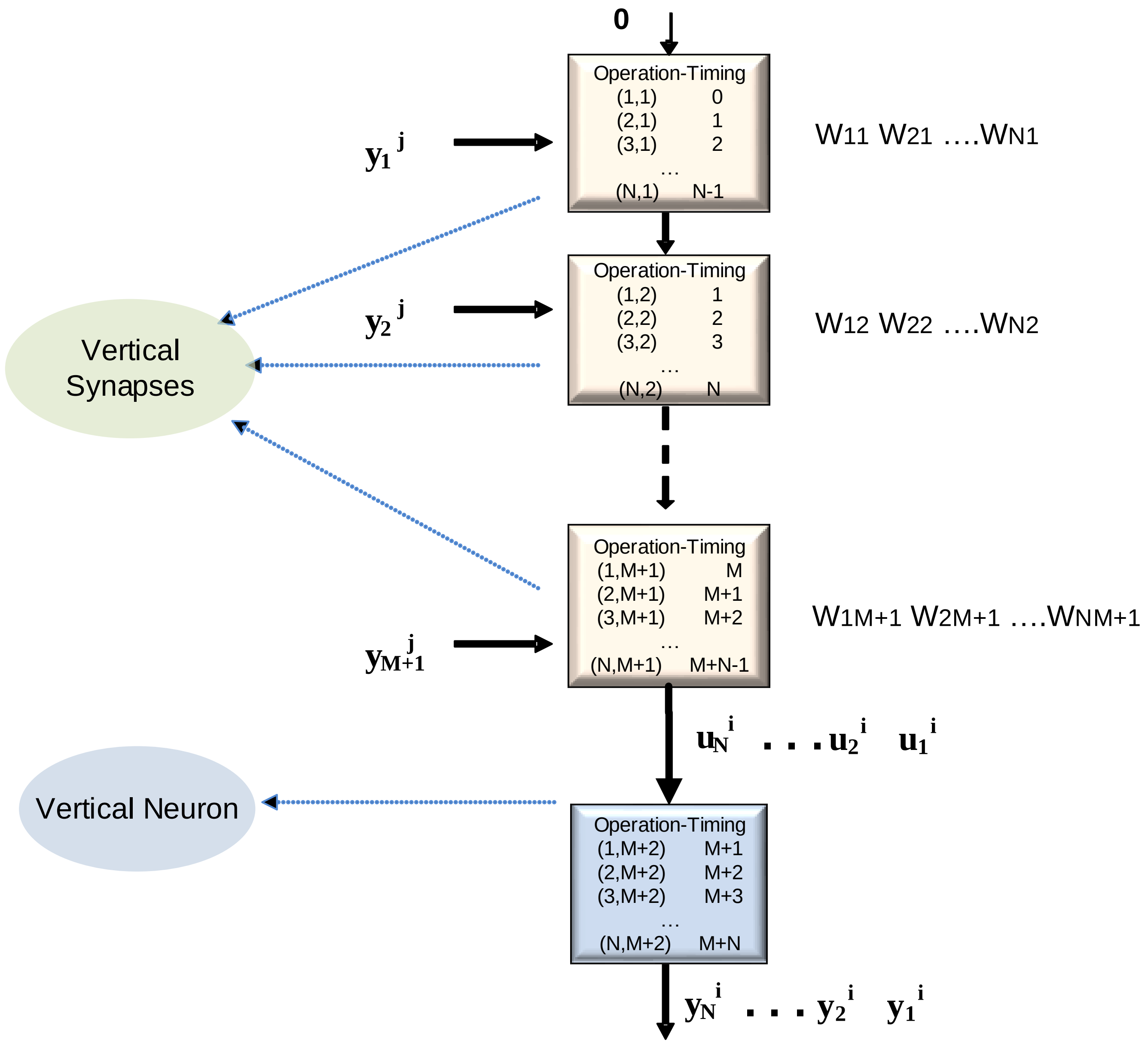


Figure 4. Systolic array encoder layer.

The fundamental characteristics of this projection in a interconnection layer are:

- The number of functional blocks, called “vertical synapses”, equals the number of neurons in the source layer. Those functional blocks basically perform the function $F = A \times B + C$; and
- The number of functional blocks (neurons) is 1. This functional block implements the activation function.

3.1.2. Decoder

An autoencoder with the same layer structure as the one shown in Figure 1 is used. In this section, the decoder layer connection between *i* (neuron source layer) and *k* (neuron destination layer) is discussed (Figure 5).

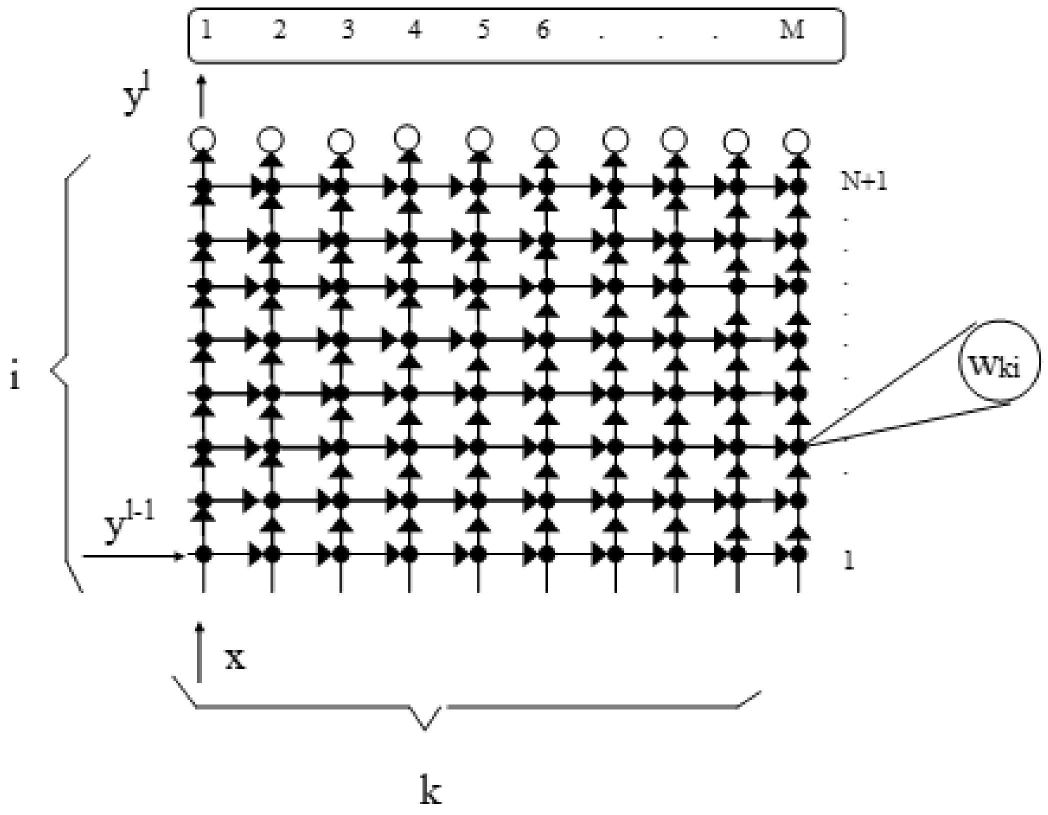


Figure 5. Decoder–layer interconnection.

1. Uniform recurrence equations.

From Equations (5) and (6), uniform iterative Equations (7) and (8) can be obtained:

$$u_k^l = \sum_{i=0}^{N+1} w_{ki}^l y_i^{l-1} \tag{5}$$

$$y_k^l = f(u_k^l) \tag{6}$$

$$1 \le k \le M,\ 1 \le l \le L,$$

where y stands for activation, w for weight, f for a the non-linear function, and i, k for the indexes in the source and destination layer.

$$layer\ l \left\{ \begin{array}{l} x^l(k,i) = x^l(k,i-1) + w^l(k,i) \cdot y^{l-1}(k-1,i) \\ y^{l-1}(k,i) = y^{l-1}(k-1,i) \\ y^l(k,N+2) = f(x^l(k,N+1)) \\ 1 \le k \le M \\ 1 \le i \le N+1 \end{array} \right\}, \tag{7}$$

and the boundary conditions:

$$layer\ l \left\{ \begin{array}{l} y^{l-1}(0,i) = f(x^{l-1}(i,L+1)) \quad \text{for} \quad \text{i} = 1 \ldots N \\ y^{l-1}(0,N+1) = 1 \qquad \text{Bias} \\ x^l(k,0) = 0 \quad \text{for} \quad \text{k} = 1 \ldots M \end{array} \right\}. \tag{8}$$

2. Δ = dependence vectors = { $(0,1)_{d_1}$ $(1,0)_{d_2}$}
3. D = {(k,i) $M \ge k \ge 1 \quad N+1 \ge i \ge 1$ }
4. Domain geometry.

The geometry of the domain is shown in Figure 6.

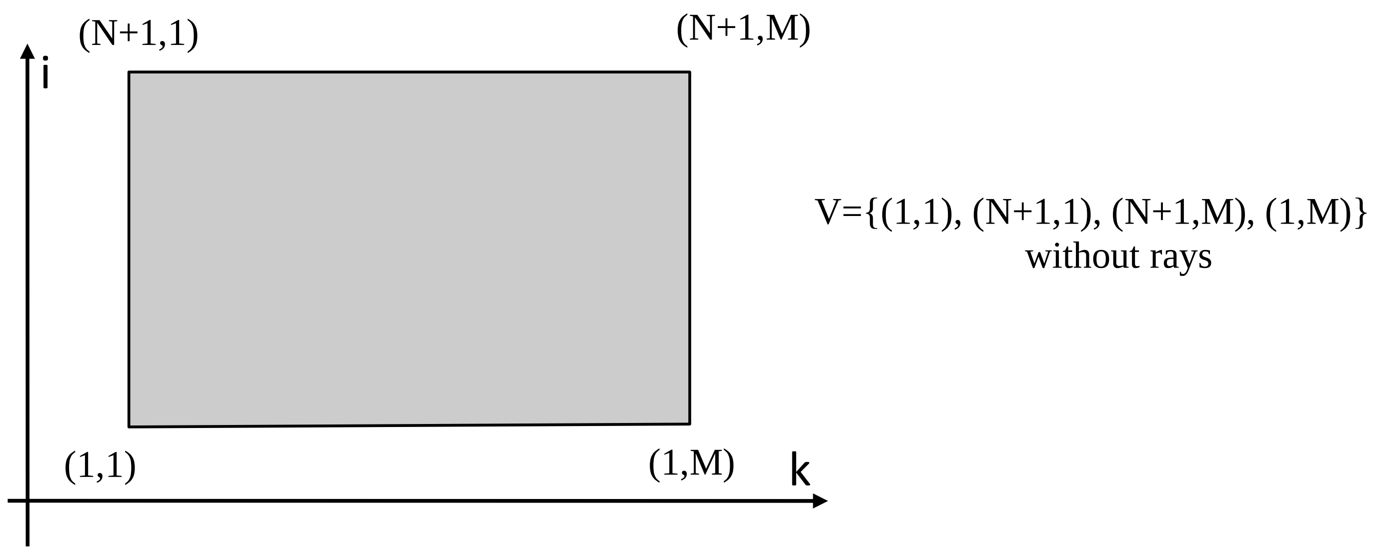
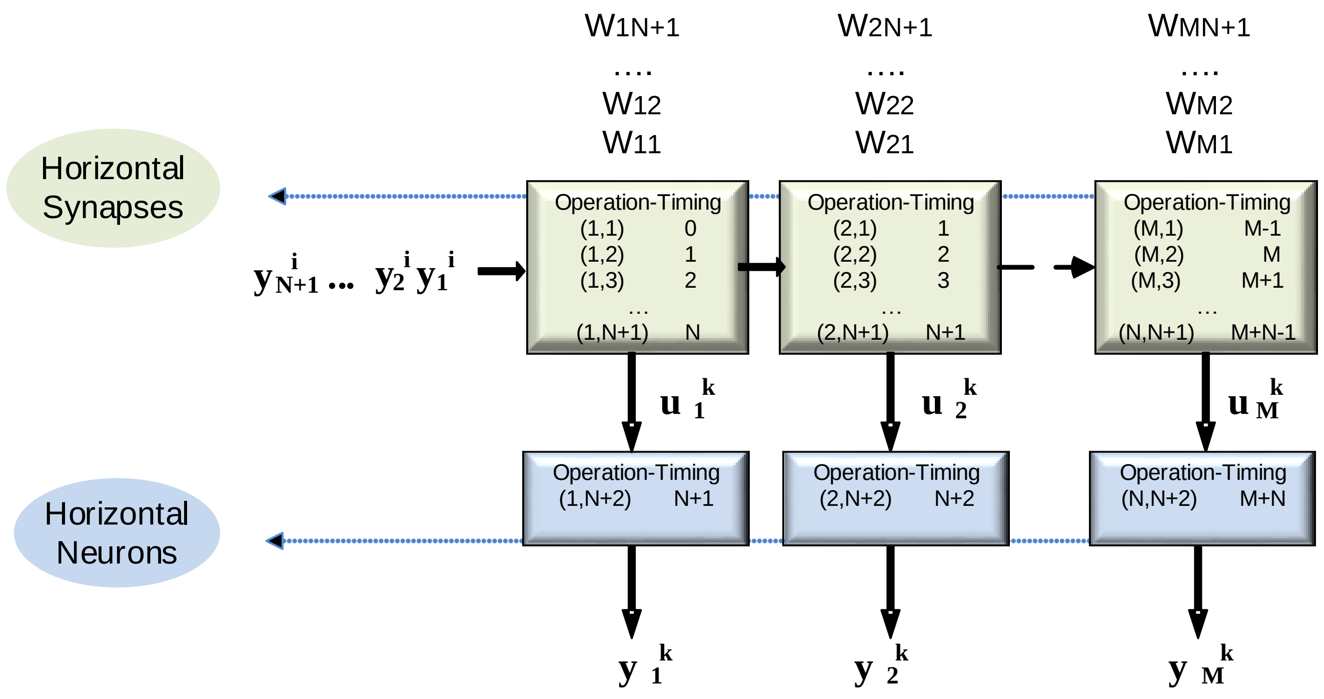


Figure 6. Domain geometry in the decoder layer.

Resulting Systolic Array

The final result of the systolization procedure with a direction (0,1) projection is the systolic array shown in Figure 7.



W1N+1 .... W12 W11
W2N+1 .... W22 W21
WMN+1 .... WM2 WM1
Horizontal Synapses
Horizontal Neurons
y^i_{N+1} ... y^i_2 y^i_1
Operation-Timing (1,1) 0 (1,2) 1 (1,3) 2 … (1,N+1) N
Operation-Timing (2,1) 1 (2,2) 2 (2,3) 3 … (2,N+1) N+1
Operation-Timing (M,1) M-1 (M,2) M (M,3) M+1 … (N,N+1) M+N-1
u^k_1
u^k_2
u^k_M
Operation-Timing (1,N+2) N+1
Operation-Timing (2,N+2) N+2
Operation-Timing (N,N+2) M+N
y^k_1
y^k_2
y^k_M

Figure 7. Systolic array decoder layer.

The fundamental projection characteristics for an interconnection layer are:

- The number of functional blocks, called "horizontal synapses", equals the number of neurons in the target layer. Those functional blocks basically perform the function $Accu = A \times B + Accu$; and
- The number of functional blocks (neurons) equals the number of neurons in the target layer. This functional block implements the activation function.

3.1.3. Alternative Architecture

The same study can be repeated by choosing the projection direction (0,1) for the encoder layer and the projection (1,0) for the decoder layer. This choice was made in order to enable an efficient communication between layers.

3.2. Conversion of the Systolic Architecture to OpenCL

The first proposed architecture (called the V-H architecture here) is shown in Figure 8.

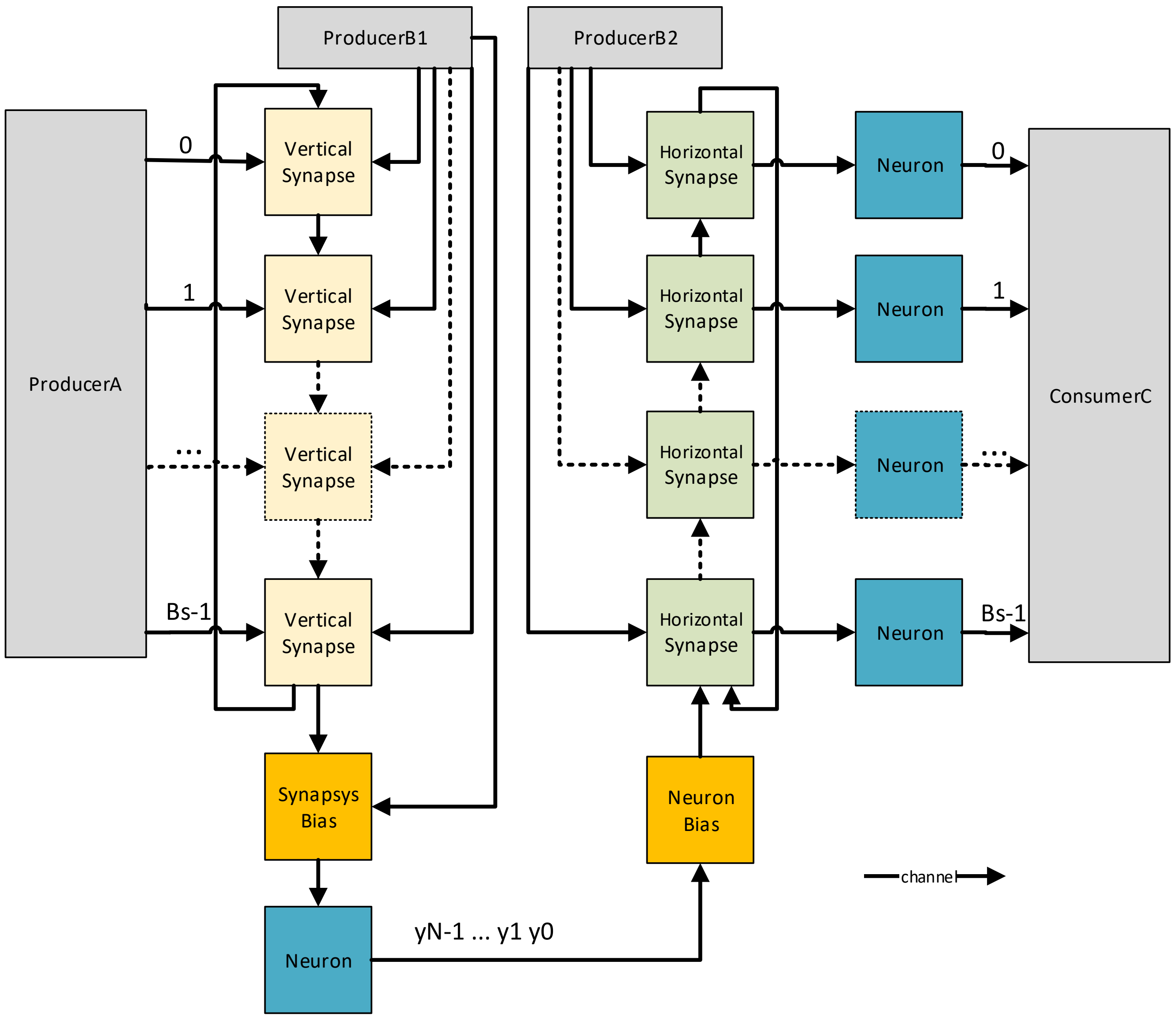


Figure 8. Systolic architecture data flow with the vertical–horizontal combination.

The second proposed architecture (called the H-V architecture here) is shown in Figure 9.

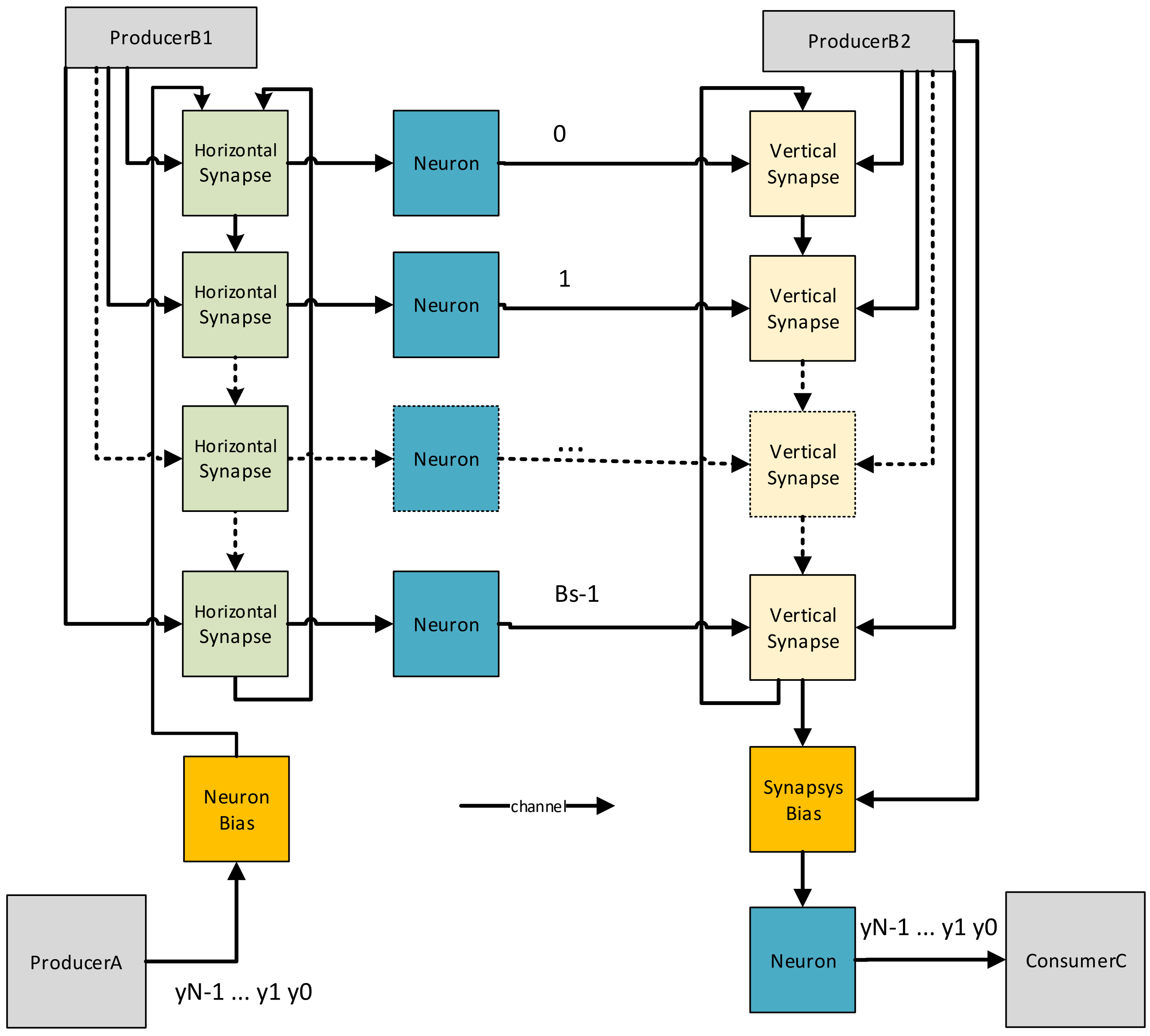


Figure 9. Systolic architecture data flow with the horizontal–vertical combination.

As already shown, these architectures are the combination of the encoder and decoder layers discussed in the previous subsection:

- All the data streams were implemented through channels. Those channels perform two important tasks: the transmission of data and control signals and the synchronization of the systolic architecture. Each processing unit (synapses and neurons) can only operate when all of the data and control channels have data available. Otherwise, these processing units are blocked while the queues (channels) are empty. Therefore, all of the processing units (synapses and neurons) act autonomously and do not need to be commanded from the host. The difficulty in this approach lies in making the systolic network adaptable to different numbers of inputs, outputs, and neurons in the hidden layer without having to recompile the OpenCL code. This task is performed by the control generation units, which distribute control commands through the associated channels. Alternative to the latter scheme, in order to improve the performance of the systolic architecture, a distributed control scheme located at each processing unit can be used instead of a centralized one. This results in a loss of flexibility. Since the processing units do not have direct connectivity to the host, it cannot modify the properties of the implemented network by means of arguments.

- The two encoder and decoder layers are constructed based on a BLOCK-SIZE parameter (Bs in Figures 8 and 9). The BLOCK-SIZE parameter is used several times to cover the number of inputs and outputs, as shown in Figure 8, or to cover the number of hidden neurons, as shown in Figure 9. In the cited experiments, BLOCK-SIZE values of 16, 32, 64, and 128 were considered, which respectively provides 32, 64, 128, and 256 floating point DSPs.
- All the operations are in single precision format.
- Theoretically, an interconnection layer systolic architecture in which the source layer number of neurons is higher than that in the target layer is more efficient when horizontal projection is used. On the other hand, a vertical projection is more efficient when the number of neurons in the target layer is larger than that in the source layer. As a consequence, the architecture shown in Figure 9 was expected to be more efficient with autoencoders whose hidden layer is smaller than the input layer. This last assumption is experimentally corroborated by means of a 640-256-640 (inputs-hidden-outputs) topology autoencoder configuration in the following sections.

3.3. RTL Refinement

3.3.1. Introduction of the Nonlinear Function inside the Kernel

Our experience in designing backpropagation (BP) algorithm accelerators [19,20] along with other works [21] indicates that one of the most effective means of achieving acceleration is usually to replace the software implementation of the hyperbolic tangent with a hardware one.

The processing units depicted in Figure 8, called neurons, necessarily include the nonlinear activation function in OpenCL based implementations. The proposed hardware implementation makes use of lookup tables (LUTs) with a memory capacity that could be implemented using the Arria 10 devices' embedded memories. An organization of $2^{10} \times 20$ is used in the proposed implementation.

For instance, when the hyperbolic tangent is used as the activation function, a direct implementation through OpenCL is highly unsatisfactory in terms of area and speed, as shown in Table 1.

As seen in Equation (9), the substitution of the hyperbolic tangent by some of its equivalent formulas can be a better implementation. Results from Expression 3 are shown in the second row of Table 1. Bearing in mind that a systolic architecture with BLOCK-SIZE = 16 would need to replicate 17 times the results shown in Table 1, the final area would be unacceptable.

$$\begin{cases} Expression1 : (e^x - e^{-x})/(e^x + e^{-x}) \\ Expression2 : (1 - (2/(e^{2x} + 1)) \\ Expression3 : (e^{2x} - 1)/(e^{2x} + 1) \end{cases} \tag{9}$$

Our results regarding the resources being used and speed performance (latency) are shown in the row labeled "RTL tanh IP" in Table 1. This approach is outstanding in terms of area and speed; moreover, it has no impact on the recall work mode of the trained autoencoder from the precision point of view.

Therefore, we attempted to insert our own Verilog RTL code in place of the current standard implementation of the hyperbolic tangent function in OpenCL (see Figure 10).

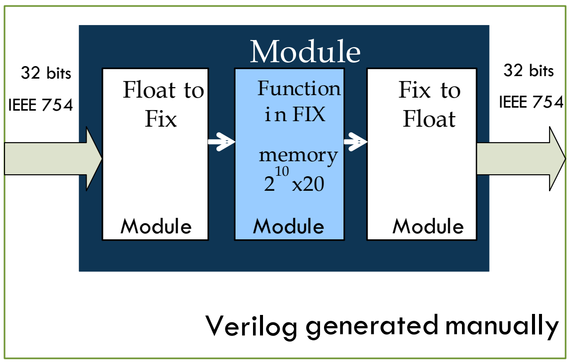


Figure 10. Structure of the RTL tanh implementation.

Table 1. Comparison between the implementations of neurons. DSP, digital signal processor.

	ALMs	**Registers**	**Memory 20 kb Blocks**	**MLABs**	**DSP Blocks**	**Latency Cycles**	**Initiation Interval**
OpenCL tanh	22,123	15,295	23	69	33	129	1
OpenCL equation	1817	177	7	22	12	64	1
RTL IP tanh	40	66	0	3	0	11	1
Without tanh	30	19	0	1	0	6	1

3.3.2. Instantiation of DSPs versus Direct Inference

The same procedure used for the implementation of hyperbolic tangent functions inside PPE neurons was applied to PPE synapses in order to achieve further refinement based on register transfer logic (RTL) libraries written in Verilog-SystemVerilog.

Results in Table 2 demonstrate that for the $F = A \times B + C$ function's implementation, the RTL code based instantiation versus OpenCL code inference has no remarkable impact in terms of efficiency. However, in the case of function $Accu = A \times B + Accu$, the amount of resources saved by using direct instantiations of horizontal synapses is remarkable (Table 3). An overall improvement in both area and performance can be observed for autoencoder implementations with 256-640-256 and 640-256-640 topologies (Tables 4 and 5).

Table 2. Comparison between $F = A \times B + C$ implementations of vertical synapses.

	ALMs	**Registers**	**Memory 20 kb Blocks**	**MLABs**	**DSP Blocks**	**Latency Cycles**	**Initiation Interval**
OpenCL equation	451	765	8	8	1	34	1
RTL IP instantiation	443	729	8	8	1	34	1

Table 3. Comparison between $Accu = A \times B + Accu$ implementations of horizontal synapses.

	ALMs	**Registers**	**Memory 20 kb Blocks**	**MLABs**	**DSP Blocks**	**Latency Cycles**	**Initiation Interval**
OpenCL equation	776	1331	8	9	1	36	6
RTL IP instantiation	374	569	8	8	1	31	1

Table 4. Comparison between two implementations of an autoencoder (256-640-256) with architecture V-H and BLOCK-SIZE = 64.

	ALMs	**Registers**	**Memory 20 kb Blocks**	**MLABs**	**DSP Blocks**	**Latency µs**	**Throughput fps**
OpenCL equation	107,155	163,976	601	1822	136	370	13,082
RTL IP instantiation	77,595	108,766	569	1822	136	236	60,716

Table 5. Comparison between two implementations of autoencoder (640-256-640) with architecture H-V and BLOCK-SIZE = 64.

	ALMs	**Registers**	**Memory Blocks 20-kb**	**MLABs**	**DSP Blocks**	**Latency µs**	**Throughput fps**
OpenCL equation	98,938	153,048	566	1516	136	400	12,891
RTL IP instantiation	68,938	98,145	566	1452	136	376	55,555

4. Performance Evaluation

The Arria 10 family was specifically introduced as the technological answer of the FPGA world (by the manufacturer IntelFPGA) to manage floating point operations and that could compete with GPUs. It is important to demonstrate in our article that the DSPs of this technological family can be handled more efficiently than the automatic inference performed by the OpenCL base compiler. This led to the development of their own libraries based on Verilog RTL code and the management of their development tools for new compilation libraries provided by the manufacturer. Of course, the development of these libraries requires being able to demonstrate with different devices of the same family that this proposed refinement is still working without problems. That is why we carried out an intensive study of the behavior of several boards with this technological family of FPGAs (Sections 4.1–4.3). It is also important to demonstrate the importance of communication between layers (observed mainly in the throughput study in Section 4.2) and host communication with the existing multikernel inside the FPGA (observed mainly in the latency study in Section 4.3).

4.1. Throughput and Latency Measurements

The process performance can be measured in two different inference ways: synchronous or asynchronous. In the asynchronous case, multiple requests are typically executed asynchronously, and the performance is measured in frames per second (fps), dividing the number of images being processed by the processing time. For latency-oriented tasks, the time to process a single frame is more important.

These measures offer comparative information between architectures and are usually provided by environments like OpenVINO as standard performance measurements.

However, it is important to clarify the following points regarding these results:

1. The autoencoder topology being used is 640-256-640. The sizes of the input and output are determined by our needs in medical imaging application where autoencoders are required.
2. Four board support packages (BSPs) for the IntelFPGA Arria 10 family of devices were used. The manufacturers of these BSPs and boards are:
   - Alaric Arria 10 (Reflex CES) and Alaric Arria 10 (Reflex CES) with the support of host pipes [22].
     - 4 GB DDR3 on-board memory.
     - PCIe Gen3 x4.
     - FPGA: Arria 10 SX: 660 K logic elements, 1688 hardened single-precision floating-point multipliers/adders (MACs), speed grade-2.
   - A10sa4 (Bittware) [23].
     - 8 GB DDR4 on-board memory.
     - PCIe Gen3 x8. PCI passthrough.
     - FPGA; Arria 10 GX: 1150 K logic elements, 1518 hardened single-precision floating-point multipliers/adders (MACs), speed grade-2.
   - PAC (Intel) [24]
     - 8 GB DDR4 on-board memory.
     - PCIe Gen3 x8.

   – FPGA: Arria 10 GX: 1150 K logic elements, 1518 hardened single-precision floating-point multipliers/adders (MACs), speed grade-2

3. BSPs describe system architectures that include a host and an accelerator connected through a peripheral component interconnect (PCI) express bus. The host sends the image matrix (each image occupies a row of the matrix) and receives the reconstructed images generated by the autoencoder (each row occupies a row of the matrix of reconstructed images).
4. In our medical imaging applications, a 10 µs throughput specification is required. This requirement must be fulfilled by a parallel set of six FPGAs. The results obtained in this work show a 11.13 µs throughput for a single FPGA. Therefore, using a proper parallelization method, the initial throughput specification could be fulfilled.
5. Networking-type solutions could also be applied to our medical imaging applications. These solutions can deal with up to 10 G data networking connections in which the fundamental communication element would be the Standard 10 G low latency MAC IP (Intel/Altera).

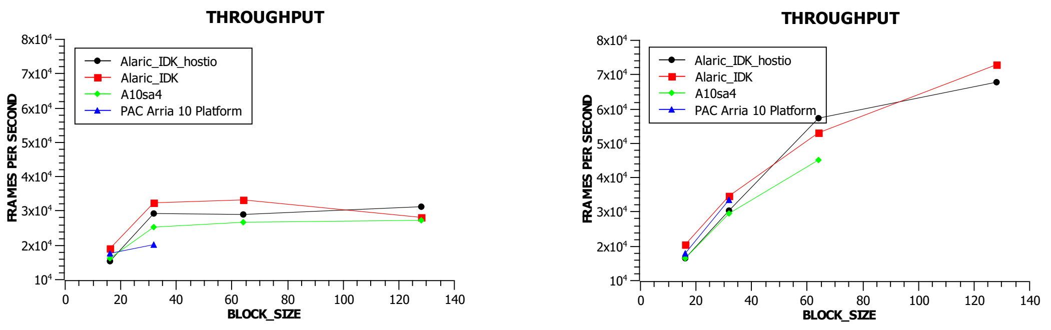
4.2. Throughput Results

A close look at the throughput results (Figure 11a,b) supports our hypothesis that the H-V architecture represented in Figure 9 is more efficient than the V-H architecture shown in Figure 8. This is a direct consequence of the underlying architectural properties inherited by the OpenCL implementation.

To further corroborate this hypothesis, an autoencoder with opposite characteristics to those required for our experiments was also implemented. The new autoencoder topology consists of a hidden layer larger than the input and output layer (256-640-256). The throughput results in Figure 12a show that efficiency with the H-V architecture is higher than with the V-H architecture for the 640-256-640 topology. As shown in Figure 12b, the efficiency with the V-H architecture is higher than with the H-V architecture for topology 256-640-256.

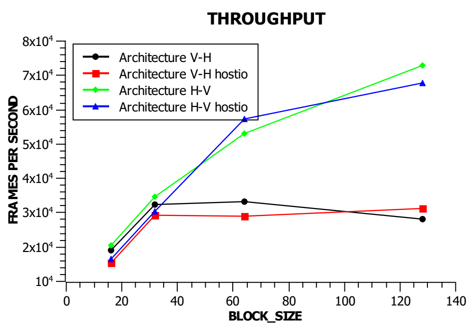
For an optimal architecture topology, a higher (BLOCK-SIZE) parameter value improves the throughput. This result was expected as the number of DSPs involved is increased by the same amount. However, efficiency per DSP shows a falling tendency, since a higher complexity decreases the maximum frequency of operation.

Regarding the different board support packages (BSPs) being used (associated with different boards), our intention was to demonstrate the adequacy and robustness of the systolic architecture rather than to establish a comparison among them. Those systolic architectures were implemented on different boards with different versions of the OpenCL compiler. In asynchronous inference mode, the use of host pipes was not decisive.

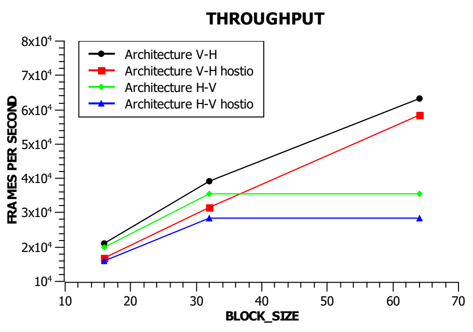


(**a**) V-H architecture. (**b**) H-V architecture.

Figure 11. Throughput with different boards.



(**a**) Topology 640-256-640.



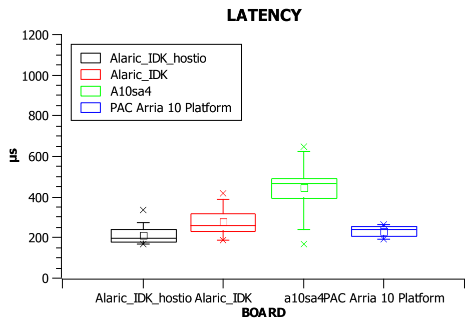
(**b**) Topology 256-640-256.

Figure 12. Throughput with the Alaric_IK board.

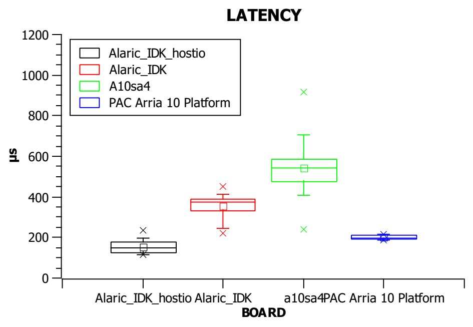
4.3. Latency Results

Regarding the latency results, the most remarkable conclusion is that such latency is hardly dependent on the architecture being used (compare Figure 13a,b) and slightly dependent on the BLOCK-SIZE parameter value (shown in Figures 14a,b–17a,b).

Communication between the host and kernels, which receive the input data and transmit the output results back to the host, has the strongest impact on the latency. Some BSPs are able to implement host pipes, which allow the lowest latencies.

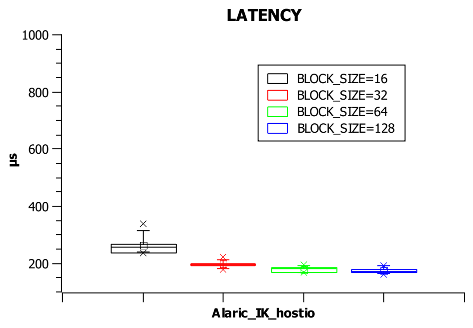


(**a**) V-H architecture.

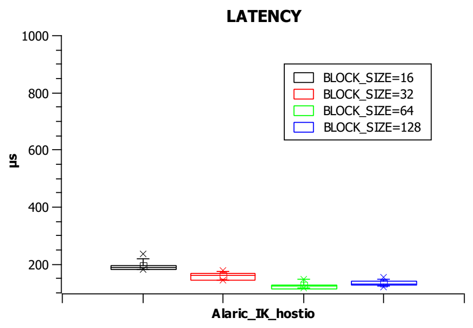


(**b**) H-V architecture.

Figure 13. Latency with different boards.

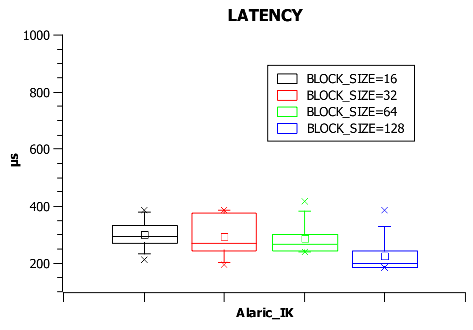


(**a**) V-H architecture.

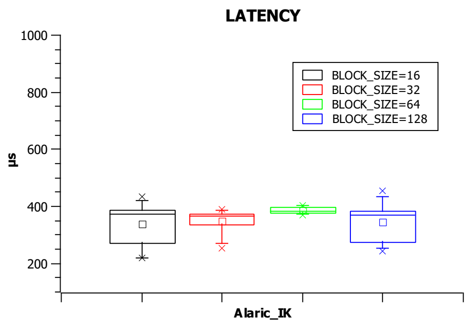


(**b**) H-V architecture.

Figure 14. Latency with the Alaric_IK_hostio board.

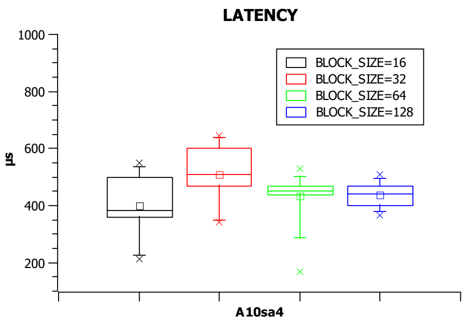


(**a**) V-H architecture.

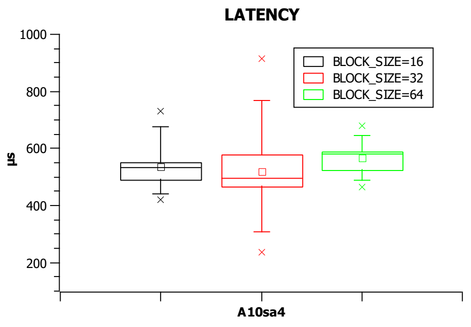


(**b**) H-V architecture.

Figure 15. Latency with the Alaric_IK board.

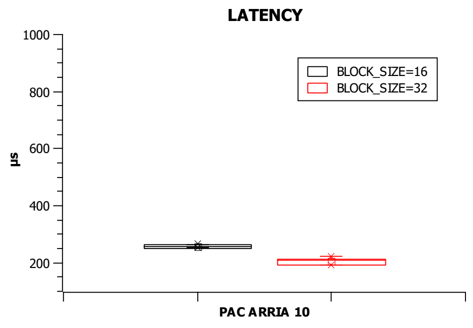


(**a**) V-H architecture.

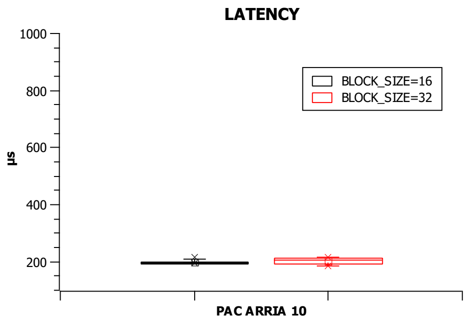


(**b**) H-V architecture.

Figure 16. Latency with the A10sa4 board.



(**a**) V-H architecture.



(**b**) H-V architecture.

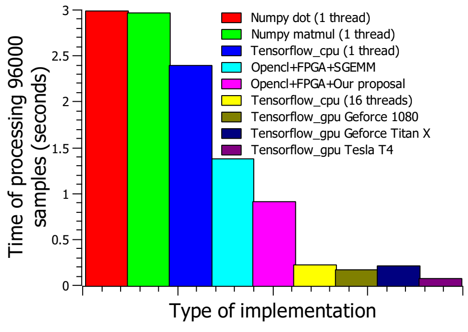
Figure 17. Latency with the PAC board.

4.4. Comparison of the Implementations

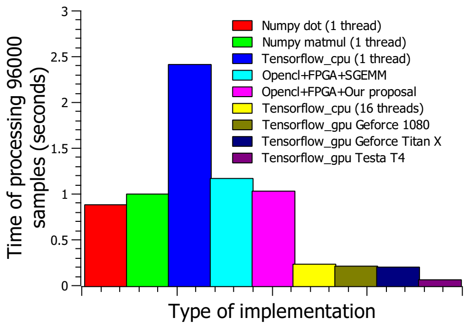
In this subsection, our autoencoder based systolic architecture processing speed is compared to other implementations: CPU, GPU, and FPGA solutions based on the single float precision general matrix multiply (SGEMM) implementation.

Figure 18a,b shows the speed performance of our proposal compared to other hardware and software implementations. The GPU based implementations are the most efficient and virtually share the same degree of flexibility with CPU based implementations. The implementation platform was Tensorflow Version 2.2.

Regarding FPGA based hardware accelerators, the differences are no longer so evident (Tables 6 and 7). Neither of the implementations reach the specified throughput (10 µs per image or 100,000 fps). However, both reach those specified constraints if we eliminate the communication times between the host and kernel. The execution time of the internal kernel of the FPGA is less than 0.96 s in both cases. The performance of the internal kernel based on single float precision general matrix multiply (SGEMM) (Table 7) is even better than our implementations. However, the computational efficiency must be taken into account. In SGEMM based solutions, the number of DSPs used is four times higher than in our systolic architecture. Therefore, its computational efficiency (GOP/s/DSP) is worse than that of our proposal.



(**a**) With the tanh function.



(**b**) Without the tanh function.

Figure 18. Autoencoder 640-256-640.

Table 6. Comparison between two implementations of autoencoder 640-256-640 with FPGA and BLOCK-SIZE = 128, processing 96,000 images. SGEMM, single float precision general matrix multiply.

	ALMs	**Registers**	**Memory Blocks (20 kb)**	**DSP Blocks**	**Frequency (MHz)**
OpenCL FPGA SGEMM	82,983	229,383	1866	1062	184
OpenCL FPGA our proposal	244,453	336,898	1528	262	230

Table 7. Performance comparison between two implementations of autoencoder 640-256-640 with FPGA and BLOCK-SIZE = 128, processing 96,000 images.

	Throughput (fps)	**Throughput Kernel (fps)**	**GOP/s**	**GOP/s/DSP**	**GOP/s/K Cells**
OpenCL FPGA SGEMM	82,332	126,315	82.74	0.079	0.996
OpenCL FPGA our proposal	90,107	111,291	72.93	0.278	0.297

Finally, there is another key point regarding both implementations: our proposed method can work with any number of input images, whereas the SGEMM implementation must work with a BLOCK-SIZE multiple number of input images (in this case, 128). In order to get the shortest time in a previously trained neural network recall phase, implementations that can only work with batches of input images should be avoided.

4.5. Discussion

The following aspects from the obtained results can be highlighted:

- GPU based implementations are difficult to beat in terms of neural network inference. We do not aim at comparing ours with such implementations, but to use them as reference points for FPGA based implementations. In terms of throughput performance for 32 bit floating point resolution implementations, GPUs beat FPGAs by a wide margin.
- With OpenCL based hardware implementations on FPGAs, a greater computational efficiency can be provided by using systolic architectures. We wanted to achieve an intermediate point between the use of HLS and HLD that would result in a flexibility close to that of HLS solutions and would improve performance. Although this was achieved, there is still a long way to go in improving OpenCL compilers, especially in the resources used for channel synthesis, accumulator synthesis, and non-linear function synthesis.
- Comparing among the FPGA implementations, we have lower performance when using HLS. We selected [13] as a reference implementation with similar characteristics as ours. This reference claims a 3.24 µs throughput autoencoder of size 784 × 196 × 784 and an 18 bit fixed point resolution. Their authors made use of a ReLU (rectified linear

unit) for the activation function implementation as a performance enhancement. In our case, a 9.47 µs throughput was achieved with a 32 bit floating point resolution using a hyperbolic tangent as the activation function.

- From the point of view of the ease of design, we cannot experimentally quantify this ease with respect to the use of HDL. We carried out HDL based implementations, as well as HLS (OpenCL) ones, though it is difficult to get a design time estimation. Moreover, there is a lack of references in the bibliography regarding this parameter. However, we claim that the creation of these systolic architectures in OpenCL with multikernel can even be automated as we already did with experiences starting from c++ and having as a destination a single-work-item-type kernel [25].

5. Conclusions

In this paper, a workflow for the implementation of deep neural networks was proposed. This workflow tries to combine the flexibility of HLS based networks and architectural control features of HLD based flows. The main tool used along the workflow was OpenCL. This system level description language allows a structural approach, as well as a high degree of control from a hardware point of view. This fact is of main importance when dealing with systolic architectures.

Following the described methodology, the performance obtained is encouraging, and the design times could be shorter provide that in the future, we could automate the generation of OpenCL from the frameworks used in machine learning. As far as the architectures are concerned, the proposed workflow could reuse classic VLSI architectural solutions or introduce new ones.

The connection between layers was shown to be a crucial issue that must be carefully studied in deep neural network implementations. A failure in choosing the right combination of projections may lead to a considerable loss of performance. The challenge is to be able to work with larger images, trying to process more than 100,000 frames per second, working with a networking-type source and destination.

Although this work focused on simple neural networks (autoencoders), our intention for future research is to extend these conclusions to interlayer communication techniques on higher complexity neural networks such as CNNs.

Author Contributions: Conceptualization, R.G.-G.; methodology, R.G.-G. and R.C.-P.; software, R.G.-G. and V.H.-B.; validation, R.G.-G. and J.M.-F.; investigation, R.G.-G.; resources, R.C.-P.; writing—original draft preparation, R.G.-G.; writing—review and editing, V.H.-B. and J.M.-F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data is contained within the article or supplementary material.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Vincent, P.; Larochelle, H.; Lajoie, I.; Bengio, Y.; Manzagol, P.A. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *J. Mach. Learn. Res.* **2010**, *11*, 3371–3408.
2. Wang, Y.; Yao, H.; Zhao, S. Auto-Encoder Based Dimensionality Reduction. *Neurocomputing* **2016**, *184*, 232–242. [CrossRef]
3. Sun, W.; Shao, S.; Zhao, R.; Yan, R.; Zhang, X.; Chen, X. A sparse auto-encoder based deep neural network approach for induction motor faults classification. *Measurement* **2016**, *89*, 171–178. [CrossRef]
4. Nurmaini, S.; Darmawahyuni, A.; Sakti Mukti, A.N.; Rachmatullah, M.N.; Firdaus, F.; Tutuko, B. Deep Learning-Based Stacked Denoising and Autoencoder for ECG Heartbeat Classification. *Electronics* **2020**, *9*, 135. [CrossRef]
5. Kunapuli, S.S.; Bh, P.C.; Singh, U. Enhanced Medical Image De-noising Using Auto Encoders and MLP. In *Intelligent Technologies and Applications*; Bajwa, I.S., Kamareddine, F., Costa, A., Eds.; Springer: Singapore, 2019; pp. 3–15.
6. Maria, J.; Amaro, J.; Falcao, G.; Alexandre, L.A. Stacked Autoencoders Using Low-Power Accelerated Architectures for Object Recognition in Autonomous Systems. *Neural Process. Lett.* **2016**, *43*, 445–458. [CrossRef]

7. Suzuki, A.; Morie, T.; Tamukoh, H. A shared synapse architecture for efficient FPGA implementation of autoencoders. *PLoS ONE* **2018**, *13*, e0194049. [CrossRef] [PubMed]
8. Coutinho, M.G.; Torquato, M.F.; Fernandes, M.A. Deep neural network hardware implementation based on stacked sparse autoencoder. *IEEE Access* **2019**, *7*, 40674–40694. [CrossRef]
9. Shawahna, A.; Sait, S.M.; El-Maleh, A. FPGA-Based accelerators of deep learning networks for learning and classification: A review. *IEEE Access* **2018**, *7*, 7823–7859. [CrossRef]
10. Venieris, S.I.; Kouris, A.; Bouganis, C.S. Toolflows for mapping convolutional neural networks on FPGAS: A survey and future directions. *ACM Comput. Surv.* **2018**, *51*, 56. [CrossRef]
11. Guan, Y.; Liang, H.; Xu, N.; Wang, W.; Shi, S.; Chen, X.; Sun, G.; Zhang, W.; Cong, J. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, 30 April–2 May 2017; pp. 152–159. [CrossRef]
12. Wei, X.; Yu, C.H.; Zhang, P.; Chen, Y.; Wang, Y.; Hu, H.; Liang, Y.; Cong, J. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In Proceedings of the 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 18–22 June 2017. [CrossRef]
13. Medus, L.D.; Iakymchuk, T.; Frances-Villora, J.V.; Bataller-Mompeán, M.; Rosado-Muñoz, A. A Novel Systolic Parallel Hardware Architecture for the FPGA Acceleration of Feedforward Neural Networks. *IEEE Access* **2019**, *7*, 76084–76103. [CrossRef]
14. Quinton, P. The Systematic Design of Systolic Arrays. In *Centre National de Recherche Scientifique on Automata Networks in Computer Science: Theory and Applications*; Princeton University Press: Princeton, NJ, USA, 1987; pp. 229–260.
15. Kung, H.; Leiserson, C. *Systolic Arrays for (VLSI)*; Department of Computer Science, CMU-CS, Carnegie-Mellon University: Pittsburgh, PA, USA, 1978.
16. Moldovan, D.I. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE* **1983**, *71*, 113–120. [CrossRef]
17. Zargham, M. *Computer Architecture: Single and Parallel Systems*; Prentice Hall: New Jersey, NJ, USA, 1996.
18. Murtagh, P.; Tsoi, A.C.; Bergmann, N. Bit-serial array implementation af a multilayer perceptron. *IEEE Proc.-E* **1993**, *140*, 277–288.
19. Gadea-Gironés, R.; Herrero, V.; Sebastia, A.; Salcedo, A.M. The Role of the Embedded Memories in the Implementation of Artificial Neural Networks. In *International Workshop on Field Programmable Logic and Applications*; Springer: London, UK, 2000; pp. 785–788.
20. Gadea-Gironés, R.; Colom-Palero, R.; Herrero-Bosch, V. Optimization of deep neural networks using SoCs with OpenCL. *Sensors* **2018**, *18*, 1384. [CrossRef] [PubMed]
21. Kumar Meher, P. An optimized lookup-table for the evaluation of sigmoid function for artificial neural networks. In Proceedings of the 2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip, Madrid, Spain, 27–29 September 2010; pp. 91–95.
22. ReflexCES. Alaric Arria® 10 SoC FMC PCIe DevKit. 2020. Available online: https://www.reflexces.com/pcie-boards/arria-10-pcie-boards/alaric-instant-devkit-arria-10-soc-fmc-idk (accessed on 30 December 2020).
23. Bittware. A10SA4 PCIe FPGA Board. 2020. Available online: https://www.bittware.com/fpga/a10sa4/ (accessed on 30 December 2020).
24. Intel. *Intel® Programmable Acceleration Card*. 2020. Available online: https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx/documentation.html (accessed on 30 December 2020).
25. Rohde, J.; Martinez-Peiro, M.; Gadea-Girones, R. SOCAO: Source-to-Source OpenCL Compiler for Intel-Altera FPGAs. In Proceedings of the FSP 2017; Fourth International Workshop on FPGAs for Software Programmers, Ghent, Belgium, 7 September 2017; pp. 1–7.